Section 7
External Communications Functions

The *External Communications* functions provide an interface between the Enhanced Traffic Management System (ETMS) and the data providers and receivers that do not implement the internal ETMS protocol. The *External Communications* functions maintain the appropriate protocols and translations necessary to interchange data between the ETMS and the systems external to the ETMS.

7.1 The External Communication Interface

The *External Communications* interfaces, the point of demarcation, and the type of information that is transferred are listed below. For simplicity the translation of external message types into NAS message types are not included; only the resultant messages are given.

7.1.1 HOST Interface

ETMS receives the full NAS collection of data from each of the 20 HOST processors. The ETMS sends Control Time messages to specific HOST processors. The demarcation point is the ETMS file server co-located at each of the ARTCCs.

7.1.2 OCS Interface

ETMS receives the entire NAS collection of data with the exception of TZ messages from the Alaskan OCS processor. This is a receive only interface. The demarcation point is the ETMS file server co-located at the Alaskan ARTCC.

7.1.3 EARTS Interface

ETMS receives TZ messages from the Alaskan EARTS processor. This is a receive only interface. The demarcation point is the ETMS file server co-located at the Alaskan ARTCC.

7.1.4 ARTS III E Interface

ETMS receives TZ messages from the Alaskan EARTS processor. This is a receive only interface. The demarcation point is the ETMS file server co-located at selected TRACON facilities.

7.1.5 ODAPS Interface

ETMS receives the entire NAS collection of data with the exception of TZ messages from the Alaskan OCS processor. This is a receive only interface. The demarcation point is the ETMS file server co-located at the New York and Oakland ARTCCs.

7.1.6 Canadian Interface

ETMS receives NAS AF/FZ/DZ/RZ/TZ data from each of the seven Canadian processing systems. This is a receive only interface. The demarcation point is the ETMS file server co-located at each Canadian Center.

7.1.7 London Interface

ETMS receives NAS FZ/DZ/TZ data from London Center. This is a receive only interface. The demarcation point is the ETMS file server co-located at London Center.

7.1.8 OAG Interface

ETMS receives weekly updates of scheduled flight plans (ETMS FS messages) from the OAG. This is a receive only interface. The demarcation point is the ETMS hub site facility.

7.1.9 ARINC Interface

ETMS disseminates flow control messages to the airlines through the *ARINC* network. The airlines also can send substitute/insert messages to the ETMS. The ETMS then sends SI Responses back as needed. The demarcation point is the ETMS hub site facility.

7.1.10 NADIN Interface

ETMS disseminates flow control messages to national and international flow control facilities through the *NADIN* network. The ETMS also receives various messages of note through this interface. The demarcation point is the ETMS hub site facility.

7.1.11 OMP Interface

ETMS translates VHF position reports from *ARINC* into ETMS TO messages. This is a receive only interface. The demarcation point is the ETMS hub site facility.

7.1.12 Weather Interface

ETMS receives raw weather data from the WSI Satellite Distribution Network. This is a receive only interface. The demarcation point is the ETMS hub site. ETMS translates the data into weather products and distributes the products to each ETMS facility.

7.1.13 ATA Interface

ETMS transmits a filtered composite of NAS and ETMS value added data to the ATA distribution system. This is a transmit only system. The demarcation point is the ATA network interface at the ETMS hub site facility.

7.1.14 ASDI Interface

ETMS transmits a filtered composite of NAS and ETMS value added data to authorized and registered airline industry related vendors. This is a transmit only system. The demarcation point is the ASDI/CDM network firewall at the ETMS hub site facility.

7.2 NAS Server

Purpose

The *NAS Server* is the front end for the *NAS Driver*. It receives messages from the driver and passes the messages to the appropriate clients. It also forwards messages from clients and/or com_server mailboxes to the HOST. All messages from the NAS network are written to a log file (i.e., //dsk04/traffic/nas_msgs.961014170002). A new file is opened once an hour.

Input

Runtime Parameters. There are two optional arguments to the *Nas.server*. The first argument is the name of the adaptation file. If none is given or it is invalid, the filename defaults to /<etms5>/nas/config/nas.server.config.

The second optional argument specifies how long to age a buffer of NAS messages before shipping it to the ETMS. The values can range from **05** to **60** seconds. The recommended interval, which is the default, is **20** seconds.

Statistical Requests.

Adaptation File Format. The first section (or line) of an adaptation file contains ARTCC code. Each data source is assigned a one-character center code. **????** Note that any blank line or any line beginning with **#** is ignored.

The second section (or line) contains the server mailbox name. This must match the name specified in the Driver configuration file. A typical line is:

/mbx/nas.mbx

The third section (or line) contains the shared region name. Short TZ and all DZ messages are written into this shared region. This allows other programs to read this data in real time. The typical line reads:

mbx/nas.region

The fourth section contains the addresses of processes to receive the buffered NAS data. There can be up to 16 entries in this and the next section. This address table must be terminated with a line beginning with the word END.

The fifth and final section is a list of addresses to be notified of various noteworthy events (users to be notified), HOST up/Down, messages timed out, messages being rejected. There can be up to 16 lines of addresses (counting the previous section) with one address per line. The format of the address is as follows:

(aaa.bbb.ccc), where aaa is site, bbb is node, and ccc is class

For example, (6.4095.9) causes errors to be sent to all occurrences of class nine on site six (all *Net.mail* on $vntsca).

Output

Status Window. *Nas.server* also maintains a status window on the screen. This window shows the current count of messages sent or received from the HOST and the status of the interface.

Hourly Log Files. All messages to or from *Nas.server*, with appropriate control information, are written into files which are created for each hour. A sample filename is //dsk04/traffic/nas_msgs.960809190000.

Processing

The *Server* program responds to statistics requests from *Net.mail*. A *Net.mail* **S0** request returns the large statistics list. This **S0** report lists the driver connection, its state, message counts and their timing. A *Net.mail* **S1** request causes a poll to be sent to the driver. The information received from the driver is reformatted by Nas.server into a simple table. A *Net.mail* **S2** request returns the clients registered to *Server* and the adaptation table of addresses to client connection types. A *Net.mail* **S3** request returns the list of messages queued to the driver.

7.2.1 Nas Server Routines and Procedures

Procedure *add_to_nas_queue* stores messages destined for the HOST/EARTS prior to their shipment. These messages currently consist of CT messages destined for the HOST processor.

Procedure *check_svr_mbox* reads the mailbox that is primarily used for data interchange between the driver and *Nas.server*. A connection request (mbx_$channel_open_mt) causes mbox_process_open to be called. A disconnect notification (mbx_$eof_mt) causes the activation of mbox_process_open. All data messages are sent to the svr_process procedure. Note that mailbox channels stop working when 2^{32} bytes of data are transferred. This procedure checks for this and unilaterally disconnects any client whose message traffic exceeds 0x37777777 bytes of data.

Procedure *check_timer* performs time oriented activities. If a buffer of messages from the HOST has aged beyond the threshold, this routine calls etms_hub_send to transfer the data. If the NOOP timeout has occurred, each mailbox client is sent a NOOP message.

Procedure *check_truncate_file* rewinds a file if it becomes larger than 65KB.

Procedure *clear_chans* closes and deallocates all active mailbox channels. This occurs on a reconfigure and on program termination.

Procedure *clear_create_nas_queues* deletes and then recreates the permanent queue of messages destined for the HOST. The filename is /traffic/nas_queue_file. It is a permanent file, and as such its contents can survive a program termination and restart. This routine creates the new queue and configures its control information.

Procedure *clear_hidden_line* erases information hidden below the horizon of the status window. This information is more complete than what is displayed but is used only for debugging purposes.

Procedure *compare_strings* compares two strings arithmetically. It returns a 0 if they match, a -1 if string one is less than string two, and +1 otherwise.

Procedure *create_log_file* creates the hourly transaction file. These files (i.e., //dsk04/traffic/nas_msgs.961014170002) contain a tracing of all message transfers between the server and the driver.

Procedure *create_nas_queue* reloads the queue file of messages destined for the HOST. This is called at initialization. If there is inconsistent control information in the file, clear_create_nas_queues is called to recreate the file.

Procedure *create_window* creates the status window. This contains the current status of *Nas.server* as well as message counts of traffic through the program.

Procedure *display_error* appends user defined text onto operating system error messages and writes the consolidated output to both stream_$stdout and into the trace file.

Procedure *display_nas_msg* decodes a packetized NAS message from the driver.

Procedure *display_nas_state* writes the current NAS state into the status window.

Procedure *display_net_error* appends user defined text to network addressing error messages and writes the consolidated output to both stream_$stdout and into the trace file.

Procedure *etms_hub_add* buffers messages from the HOST/EARTS prior to sending them to the ETMS system. If there are more than 6,000 bytes in the buffer, etms_hub_send is called to transmit the buffer. Otherwise, the four-byte timestamp, the one-byte center code, and then the received message are added onto the buffer.

Procedure *etms_hub_init* sets up a buffer to the ETMS. It sets the eight-byte password, the four-byte timestamp, the center code, the SQ message code, and the four ASCII digit sequence number. The buffer creation time is also reset.

Procedure *etms_hub_send* sends the ETMS buffer to all specified addresses, those that were read from the adaptation file at initialization. There can be up to eight destination addresses per message and the appropriate number of messages are sent to notify all the recipients. After sending etms_hub_init is called to reinitialize the buffer.

Procedure *extract_sequence* circumvents PASCAL type checking and extracts the message sequence number from a binary record.

Procedure *format_window_header* clears the display window and displays the appropriate information about the current *NAS Server* state.

Procedure *increment_message_counts* determines the index to the counters based upon message type. The appropriate counters are then augmented.

Procedure *initialize* performs the steps necessary to configure *Nas.server*. It creates all the mailbox and control tables and creates the server mailbox. While performing these operations, it calls the following initialization routines:

- initialize_database
- make_region
- create_nas_queue
- net_open
- etms_hub_init
- create_window

Procedure *initialize_database* creates the necessary directories and reads the adaptation file. Initialize_database begins by calling get_etms_path to determine the version 5 root directory,

currently /etms5. If it fails, the program terminates. This procedure then creates the /sio_files and /<etms5>/nas/trace directories and subsequently creates the trace file /<etms5>/nas/trace /nas.server. This procedure then opens the adaptation file (specified by program argument one) or the default /<etms5>/nas/config/nas.server.config.

This procedure reads the adaptation file and obtains the center code, the server mailbox name, the shared region name for TZ messages, and all the addresses that will receive messages from this program. The format of this file is specified in a separate section of this document.

Procedure *kill_nas_driver* terminates the process named *nas.driver*. This routine is called when a message times out during transmission to the HOST. There is a check to see that the driver is not terminated too frequently. Nodescan then restarts the driver.

Procedure *log_outbound_message* writes information into the hourly log file.

Procedure *make_region* creates a shared memory region for TZ messages. *Nas.server* writes them into the region; anyone can read them.

Procedure *mbox_process_eof* handles a mailbox disconnect message. This procedure releases the mailbox resources and clears all the tables that reflected the connection.

Procedure *mbox_process_open* accepts mailbox connections. The typical conection is the driver. If the connection type is not NAS, DISPLAY or PRINTER, the connection is refused. Otherwise the channel data is configured, and an accept message is sent to the mailbox.

Procedure *net_open* opens the connection to the network addressing message switching system. The class is NAS.

Procedure *net_poll* looks for messages from the network addressing message switching system. If it receives a fatal connection error, it closes and reopens the connection. Otherwise, all valid messages are sent to the net_process_message procedure. Returned messages are logged to the screen and into the trace file. All messages are read on each procedure call.

Procedure *net_process_message* determines the message type and performs the appropriate action. The following table illustrates the processing:

- net$_t_give_status_lev_1 causes a call to process_s1
- net$_t_give_status_lev_2 causes a call to process_s2
- net$_t_give_status_lev_3 causes a call to process_s3
- all other stat requests cause a call to process_stats
- net$_t_reconfigure causes a call to initialize_database, with acknowledgment to the requestor.

- ct_output_message_code (16435) causes a message to be sent to the driver (this is a CT message).

Procedure *process* is a simple loop which is gated by event counters. On each cycle it checks to see whether a new hourly file should be created, checks for sending and receving timeouts, and calls check_timer, check_svr_mbox, net_poll, and send_queue.

Procedure *Process_driver_stats* handles the statistical response from the driver. Earlier, one or more users sent an S1 command to the server. The server then sends a statistical request to the driver. The response is encoded into an ASCII message, framed for ETMS transmission, and sent to all users who are awaiting the S1 response.

Procedure *Process_driver_status* processes status type messages from the driver. In most cases these are unsolicited responses. The following table lists the various status messages and the processing that results:

- ipc_$status_ok signifies that the message was sent to the HOST. This results in a call to Process_message_sent and if there are queued messages, a call is also made to send_queue.

- pc_$status_failed signifies that the message was refused by the HOST. This results in a call to Process_message_aborted and if there are queued messages, a call is also made to send_queue.

- ipc_$status_ready signifies that the HOST circuit is now available. All users are notified and send_queue is called if nothing is outstanding to the driver.

- ipc_$status_alive signifies that the driver is now operational. Users are notified and send_queue is called if nothing is outstanding to the driver.

- ipc_$status_notready signifies that the driver is in a not ready state. Users are notified by a call to send_to_users.

- ipc_$status_noline signifies that there is no HOST connection. Users are notified by a call to send_to_suers.

- ipc_$status_msgrejected signifies that HOST rejected the message. This results in a call to Process_message_aborted and if there are queued messages, a call is also made to send_queue.

- ipc_$status_noresponse signifies that the HOST network did not send a response to the last message. This results in a call to Process_message_aborted and if there are queued messages, a call is also made to send_queue.

- ipc_$status_dcpfail signifies that the DCP microcode failed. This means that the driver needs to be reloaded. Users are notified by a call to send_to_users.

- Any other status message is unexpected; it will be logged and otherwise ignored.

Procedure *Process_message_aborted* handles the case where the connected network does not accept the sent message (the format was wrong, the line was down, the line was not turned around, etc.). If there is no outstanding message to the driver, there is no processing. Otherwise, the queued message is updated. If the retry count has not been exceeded, this routine does nothing. Otherwise, the message is removed from the queue, users are notified by a send_mail_notice and the events are logged into the hourly file.

Procedure *Process_message_sent* handles the case where a message is successfully transferred to the HOST. It removes the message from the queue and logs the event into the hourly file.

Procedure *Process_nas_rcv_timeout* handles the case where no data is being received for a while. If there is no error window currently displayed, it calls create_error_window to make one. Updates are made to the error window once a minute; if a minute has not passed, nothing is done. Otherwise, the window receives an updated *no input NAS data for xx minutes* message, and a copy of the message is sent to all users listed in the adaptation file.

Procedure *Process_nas_xmt_timeout* handles the case of no acknowledgment from the driver on a message to NAS. There is a safety check to see that the driver has a message; if it does not, nothing is done. Otherwise, an entry is made into the hourly file, a message is sent to all users listed in the adaptation file, and kill_nas_driver is called to terminate the driver. This covers the case where a driver has problems.

Procedure *Process_s1* processes the user request for driver statistics; the user entered an s1 command to *Net.mail*. This procedure adds the user's address onto a pending queue and sends a statistics request to the driver. When the response goes back later, it is processed by process_driver_status, which formats and sends the ASCII response back to the requesting *Net.mail.*

Procedure *Process_s2* processes the user request for the message distribution information; the user entered an s2 command to *Net.mail.* This routine formats the ASCII report of the people to receive system notices, the address mapping table, the process registration list, and the auto distribution list. The appropriate statistical information is also recorded.

Procedure *Process_s3* processes the user request for the display of all pending messages to HOST; the user entered an s3 command to *Net.mail.* The message queue is traversed and appropriate information is extracted and formatted into ASCII messages which are then sent to the requesting *Net.mail* process.

Procedure *process_stats* processes the user request for statistics; the user entered an s0 command to *Net.mail*. This procedure formats the appropriate statistics and returns the ASCII report to the user.

Procedure *record_status_change* writes driver status changes to the hourly log files as well as into the display window.

Procedure *region_ftm_add* adds the DZ and TZ (if the TZ is smaller than 32 bytes) into the shared region.

Procedure *send_buffer* sends messages to a specified server mailbox channel.

Procedure *send_ok_to_driver* sends an acknowledgment that the last message was received properly to the driver.

Procedure *send_queue* sends the first message on the output queue to the driver. If there already is a message outstanding to the driver, nothing on the queue, no driver connected or if the *Arinc* network is down, this routine does nothing.

Procedure *send_to_clients* distributes noteworthy event messages to all users specified in the adaptation file.

Procedure *send_to_driver* begins by checking to see if the message is the type to be queued (i.e., an outbound message) and, if so, calls add_to_nas_queue to queue it. If there already is a message outstanding to the driver and the message is queued; return. Messages that are not eligible to be queued are control type messages and should be sent immediately.

Procedure *svr_Process* processes the mailbox that the server created. At this point in time, the only client on this mailbox should be the driver. The following processing is performed on messages from the driver channel of the mailbox:

- Record the time of receipt and clear any *no messages received* flags, and call send_to_users to state that data is again being received.
- If the message from the driver is of type ipc_$stats, call process_driver_stats.
- If the message from the driver is of type ipc_$msg, send an acknowledgment to the driver by calling send_ok_to_driver.
- If the message from the driver is of type ipc_$status, call process_driver_status.
- Any other message type is displayed, then ignored.

The message type of **ipc_$msg** is further processed by logging it into the hourly file and if a client is connected to the mailbox, the message is sent to the process. The processing is completed by calling reformat_nas_msg and etms_hub_add to buffer the data.

Input messages from channels that are not the driver are scanned and if the first two bytes are a valid NAS message code (for example, CT), the message is sent to the driver by calling send_to_driver.

Procedure *update_screen_count* increments the counters for messages read from or written to the driver. This information is displayed in the status window.

Procedure *write_header* writes uniform headers into the hourly files before any information is written.

7.3 NAS Collator Function

Purpose

The *NAS Collator (NASC)* consists of one process (see Figure 7-1). *NASC* receives all NAS messages from the NAS.server, ARTS.server, London, and DOTS. *NASC* validates the format of each incoming message and collates the messages. *NASC* also filters the data (fields 3 and 11) if the filter command is used when invoking *NASC* or if the filter option is specified in the configuration file. *NASC* then sends the data to the specified destinations in the configuration file.

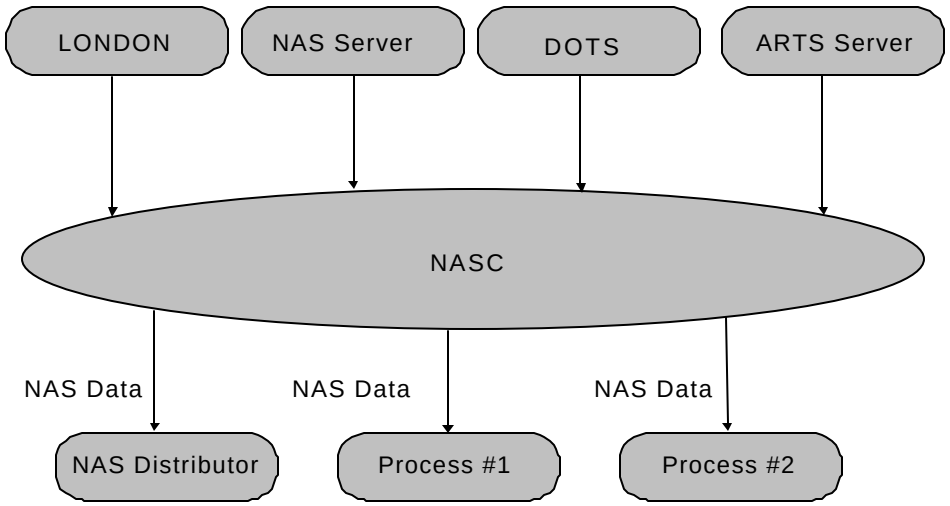

Figure 7-1. Data Flow of the NASC Process

Execution Control

NASC is a continuous process that is invoked by the run-time support process called *Nodescan*.

Input

NASC expects the name of the configuration file as its first argument for initialization. If no configuration file is provided at startup, the software defaults to:

/etms5/nas/config/nasc.config

There are two parts to the configuration file, the destination address list and the site list table.

The destination address list is a list of addresses used to send out messages from *NASC*. The addresses in the list follow the conventions for *Network Addressing* addresses. Following the address, a switch to send the data filtered or unfiltered is included; if omitted, the default for filtering is as specified in the invocation of the *NASC*. Note that the address list ends with a terminator line, consisting of the word END. The format of each entry in the list is:

(<site id>, <node id>, <class>) <filter>

There can be a maximum of sixteen destination addresses with a filtered or unfiltered option specified on each. The filtered or unfiltered option can be omitted from the configuration file, in which case the filter command line option is used. If neither the configuration file or the command line have a filter/unfiltered option used, the default of unfiltered is used.

The format of the site list configuration file is as shown below:

<NAS symbol > <ETMS site address> < ETMS site ID> <ASCII site name>

NASC expects to receive data from these sites. The ASCII site name is included in the configuration file for informational purposes within the configuration file; it is not used by *NASC*.

NASC accepts up to two arguments at initialization. The format to invoke *NASC* is:

<NASC executable> <NASC configuration file> <filter>

The second argument, which is the filter switch, is optional. This switch filters data, specifically field 3 and field 11 in the data messages. The destination addresses in the configuration file can also specify a filter option. The relationship between the command-line filter switch and the switch(es) in the configuration file is that the command-line filter sets the default mode, but the filter/unfilter value on the configuration file line will override the default for that site. If filter is not specified on the command line, the default is unfiltered.

Output

NASC forwards data either filtered or unfiltered to its list of destination addresses.

Processing

The *NASC* is a process that receives NAS data from the NAS.server, ARTS.server, London, and DOTS. The data flow of messages is shown in Figure 7-1. *NASC* forwards the data as shown in the above figure to its specified destination addresses. The destination process is most often a NAS.dist on a particular string, but it can be any process.

NASC validates the format of the data and filters the data if specified from the command line and/or the configuration file. The valid format for message data consists of an eight-byte security code, a four-byte time stamp, a one-byte center code, then the message. The first message in each buffer must be an SQ, followed by a four-digit sequence number. The filtering process removes asterisks and extra spaces from the messages. The filtering process also modifies the contents of field 3 and field 11 of the message data, per NRP rules.

NASC collates the data and responds to *Net.mail* stats level **0** command (**s0**) requests. An **s0** request gives information on the number of messages received and sent, and a breakout of the types of message received from each of the sites in the configuration file site list. There are two statistical tables that are kept by *NASC* which are displayed. The first table is called *Messages this Hour*. It displays the number of XX (unknown messages, e.g., SQ and TO messages), TZ, AF, AZ, DZ, FZ, RZ, UZ, and BZ messages that were received per site within the last hour of operation. The second table is called *Total Messages*. This displays the same information as the first table, except that it shows all of the messages that have been received since the invocation of the *NASC* process. In addition, this table also displays the number of missed blocks of data, the last block of data received, the number of times that the remote site restarted and the number of seconds since the last block of data was received per site. The SQ also provides information about the name of the adaptation file, number of unfiltered addresses, and number of filtered addresses. Stats level **s1** through **s9** are not supported by *NASC*.

NASC supports a reconfiguration command through *Net.mail*. The format of the command is:

<reconfigure> < address of the process> <name of the configuration file>

Error Conditions and Handling

The *NASC* program displays error messages under the following circumstances:

- Unable to open adaptation file <adaptation file> - This error message is displayed in the trace file when *NASC* is invoked without specifying the configuration file, its mandatory argument. The program aborts if it is unable to open the adaptation file.

- There is no directory structure specified - This error message is displayed in the *NASC* process window when the etms5 directory is missing.

- Unable to open adaptation file<adaptation file>- This error message is also displayed in the trace file when the configuration file does not exist or when it is missing from the ETMS5 tree.

- Reconfigure command sent to <NASC process> - This message is displayed when a reconfiguration is attempted using an invalid configuration file. Since the reconfiguration cannot be successfully completed, a confirmation message is never displayed.

- IOS_GET_ERROR - This error message is displayed if NASC is unable to read a line in the adaptation file.

- Invalid termination of address field. A line with `end' is the terminator - This error is displayed if the addresses specified in the configuration file are not terminated with END .

- Port is filled Msg lost - This error message is displayed in the trace file when a ERR_NET_T_PORT_FILLED error is generated.

- ***Error in adaptation , line<line in file> - first element must be one character - This message gets printed to the trace file if the first element in the site list table is invalid.

- ***Error in adaptation , line<line in file> - second element must be three characters - This message is printed to the trace file if the second element in the site list table is invalid.

- ***Error in adaptation , line<line in file> - third element must be numeric - This message is printed to the trace file if the third element in the site list table is invalid.

7.4 NAS Distributor Function

Purpose

Nas.dist accepts data from various data providers and distributes the data to registered clients as shown in Figure 7-2.

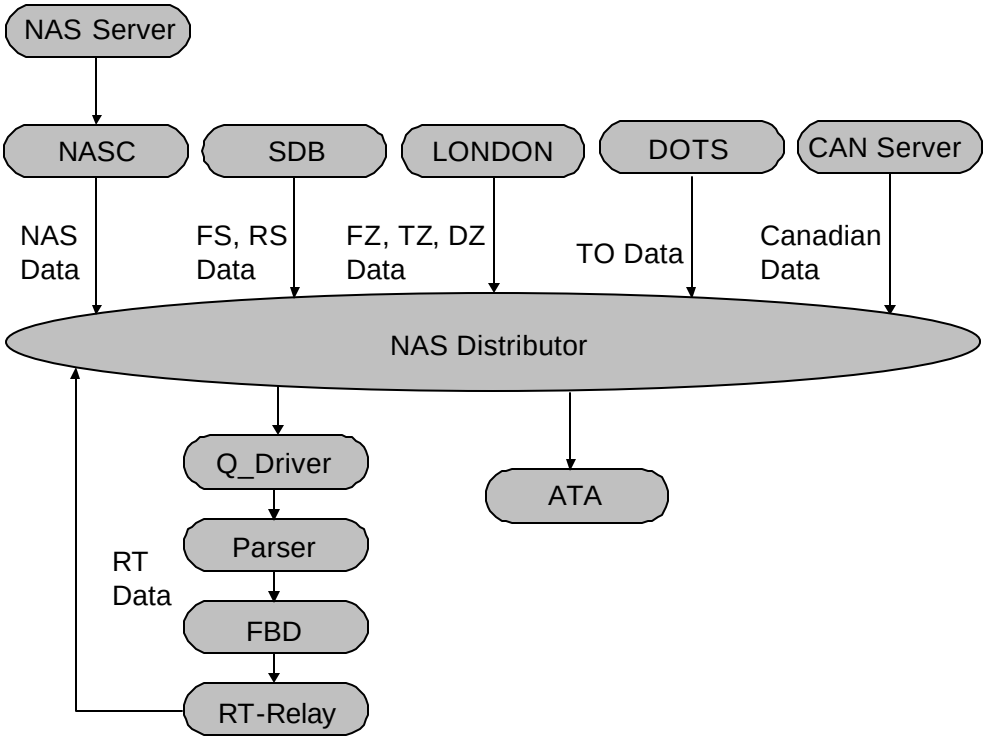

Figure 7-2. Data Flow of the NAS Distributor Process

Execution Control

Nas.dist is a continuous process that is invoked by the run-time support process called Nodescan. The format to invoke *Nas.dist* is:

nas.dist <nas.dist configuration file>

Input

Nas.dist requires the name of the configuration file as its only argument for initialization. The configuration file contains the list of acceptable clients to *Nas.dist*. The format of the configuration file is:

<Site> <Class> < Optional field for filter switch>

Site refers to the site on which the client process will be running, and *Class* refers to the class of the process. If a process tries to register to *Nas.dist*, and if it is not in the configuration file, the registration will be rejected.

Nas.dist receives data inputs from various data providers as listed in Table 7-1.

Table 7-1. Nas.dist Data Providers

Message Type	Description	Data Provider
NAS	Contains DZ,FZ,AF,AZ,BZ, UZ and RZ messages	NDA,NASC
FS/RS	Flight scheduling messages	SDB
TO	Oceanic Tracking Report	DOTS
RT	Route Message	RT-Relay
NAS	Flight position and flight plan data for Canadian flights	Can.Server
NAS	Contains DZ, FZ and TZ data only	London

Note that *Nas.dist* processing is determined by the class of the sender of the message. For example, if the data provider was DOTS, *Nas.dist* assumes that all the data coming in is of type tomsgs.

Output

Nas.dist formats the data and ships the appropriate data to registered clients. The registered clients can request any of several services from *Nas.dist* as listed in Table 7-2.

Table 7-2. Nas.dist Services

Valid Services	Description
allmsgs	All Messages
no-rte	All messages except rt messages
fsmsgs	FS-RS messages
tomsgs	TO messages
nasmsg	NAS messages
rtmsgs	RT messages
nortfs	no RT, FS or RS messages

Nas.dist writes the data that is sent out to a raw file. A raw file is a time-stamped log file created every hour, and the raw data (data before any validation or formatting) read by *Nas.dist* is written into it.

Processing

Nas Distributor is a process that receives data from various providers. *Nas Distributor* reads the data shown in Figure 7-2. It requires that the message has 8 bytes of security appended to the beginning of the message.

As seen above, *Nas.dist* receives various data. It receives this data and formats it before sending it to the clients. *Nas.dist* validates the time stamps of the data using the following rules:

(1) If a message is an AZ and time is more than +/- 30 minutes old, *Nas.dist* corrects time in the message by setting it to current time.

(2) If a message is a TZ and time is more than +/- 10 minutes old, *Nas.dist* drops the message.

(3) If time is more than +/- 7 minutes old for all other messages, *Nas.dist* corrects time in the message by setting it to current time.

If the filter switch is set in the configuration file, *Nas.dist* filters military flights from the data sent to the registered clients.

Nas.Dist responds to stats level **S0**, **S1**, and **S2**, where **S0** and **S1** requests return the same information. The **S0** and **S1** conveys information on the number of bytes sent, bytes received, and bytes rejected. It also provides information about the name of the adaptation file, number of registered clients, and number of filtered clients. **S2** gives information about the contents of the adaptation file, number of messages with validated time stamps, and number of messages lost due to bad time. Stats level **S3** through **S9** are not supported by *Nas.dist*.

Nas.dist supports the reconfiguration command through *Net.mail*. The format of the command is:

<reconfigure> < address of the process> <name of the configuration file>

Error Conditions and Handling

Nas.dist program displays error messages under the following circumstances:

- NO ADAPTATION FILE SPECIFIED! - ERR_PGM_BADARGS - This error message is displayed when *Nas.dist* is invoked without specifying the configuration file, its mandatory argument.

- Using /ETMSPATH file, cannot use /etms5 for ETMS directory. COULD NOT OPEN TRACE FILE OPENING ADAPTATION FILE! - ERR_FILE_NOTOPENDATE = 01/06/1997 TIME = 16:21:12 **ERROR** ERR_FILE_NOTOPEN - This error message is displayed when the ETMS directory is missing.

- COULD NOT OPEN TRACE FILE - This error message is displayed when trace directory is missing.

- OPENING ADAPTATION FILE! - ERR_FILE_NOTOPEN - This error message is displayed when the configuration file does not exist or when it is missing from the ETMS tree.

- COULD NOT OPEN CONFIGURATION FILE <filename> - This error message is displayed in the window where *Net.mail* is running, when user tries to reconfigure *Nas.dist* with a non-existent configuration file.

- REJECTING REGISTRATION: INVALID SERVICE COUNT - This error message is displayed when the service count exceeds NET_MAX_SERVICE_NEEDED

- REJECTING REGISTRATION: INVALID SERVICE - This error message is displayed when the service = NET_T_MSG_REG BAD

- INVALID ADDRESS FOR CLIENT REJECTING REGISTRATION: INVALID ADDRESS - This error message is displayed when the client is not found in the client table

- INVALID MSG CODE - This error message is displayed when *Nas.dist* receives a bad message code from the user

- CURRENT TIME<time> AZ WITH BAD TIME:<time> - When timestamp on an AZ message type is later than current time or over 30 minutes old, this message is logged in the trace file.

- CURRENT TIME:<time>WITH BAD TIME:<time> - When timestamp on a TZ message type is 10 minutes off, *Nas.dist* discards this message and this is logged into the trace file.

- CURRENT TIME:<time>MESSAGE TYPE:<message type> WITH BAD TIME: <time> - For all other message types which are +/- 7 minutes off of the current time, is reset to current time and this is logged into the trace file.

- COULD NOT CREATE RAW FILE - This message is displayed when *Nas.dist* fails to create a raw file

- Error trying to create traffic file dir - This message is displayed when *Nas.dist* fails to create traffic file directory

- Error trying to access traffic file dir - This message is displayed when *Nas.dist* fails to access traffic file directory

- MESSAGE FROM <message type> DID NOT MATCH SECURITY - This message is displayed when the first 8 bytes of the message read does not match the 8 bytes of security.

- MESSAGE FROM UNACCEPTABLE CLASS - This message is displayed when *Nas.dist* reads data from a process with a class not defined in its configuration file.

- ERROR : PREVENTED SENDING MSG - This error message is displayed when *Nas.dist* fails to send data to the registered clients because of port filled conditions.

- INVALID NAS MSG SIZE - This error message is displayed when *Nas.dist* does not find 8 bytes (4 of byte count, 4 of time) of control information and at least one line of data (assume 10 bytes) after 8 bytes of security inside the buffer containing data to be uncompressed.

- INVALID BYTE COUNT - This message is displayed when the byte_count of data in the buffer is less than 10 or when it is greater than NET_MAX_DATA_BYTES_IN_PACKET

7.5 Offshore Message Processor

Purpose

The *Offshore Message Processor* (*OMP*) is the successor to the original Dynamic Oceanic Tracking System (DOTS) that was used previously by the ETMS. The *OMP* was developed by following the processing rules as specified by the ODAPS system requirements.

The *OMP* is responsible for obtaining international position reports from the *ARINC* network interface and translating them into the ETMS TO oceanic position reports.

Design Issue

The *OMP* is open system compliant. If the *OMP* must connect to an AEGIS compatible *node.sw*, the interim bridge process must be used.

Input

The *OMP* uses three input files. The first file is its configuration file, the second file is the fix to latitude/longitude mapping table, and the third is the file of parsing tokens.

Sample Adaptation File. A sample adaptation file is as follows:

/etms5/dots/config/dots.config

Uncomment if new map file NEW

```
# distribution list - typically all nas.dist addresses
# it lists which addresses will receive messages from this program
#
# remaining valid lines (up to 16) : data destination list; up to 16 addresses
#                                    (xx.xx.xx) format - note no invc_num or subaddress
#                                                       (site.node.class) in decimal digits
#                                                  e.g. (6.5.39) the () and . are required !
#

(6.4095.82)
(7.4095.82)
(8.4095.82)
(10.4095.82)
(109.4095.82)
```

New/old map file

Distribution list

Fix to lat/lon table.

/etms5/dots/fix.dat

```
ABARR 411130N 735906W N
ABBN 272306S 1530706E N # From enroute supplement
ABBYS 402823N 745935W N
ABCOL 470942N 1614012E N -MVAR 3W # From DACS 6/25/92
ABE 404335N 752718W N
ABETS 360500N 1442500E N # From DACS 6/25/92
ABH 351051N 1382333E N
ABNER 343607N 1354208W N # From DACS 6/25/92
ABR 452503N 982206W N
ABSOL 271814N 1505911W N # From DACS 6/25/92
ABULO 353800N 75112W N # From Jeppesen AT1
ABZUG 392545N 755845W N
ACERT 172006N 1464430E N # From LI 4-4-91
ACK 411654N 700138W N
ACK 541000N 100000W N # Gander messages sometimes misspell ACKIL as ACK
ACKER 555012N 690506W N # Canadian HAC 4
ACKIL 541000N 100000W N # From John Paeper
ACLIF 395100N 750000W N
ACMEE 220148N 750018W N # From DACS 6/25/92
ACO 410628N 811206W N
ACONT 401548N 1494906E N # From DACS 6/25/92
ACORA 134000N 673000W N # From DACS 6/25/92
ACOVE 421405N 740156W N
ACR 351721N 1411219E N
ACRAM 420239N 734320W N
ACRON 133720N 1430144E N # From DACS 6/25/92
```

Parsing tokens file.

/etms5/dots/dots_smi

AEP 01 POSITION WITH WEATHER
AGM 07 MISCELLANEOUS AIR/GROUND MESSAGE
AID 36 AIRBORNE INSTRUMENTATION DATA SYSTEM
ALR 05 ALERTING MESSAGE
ARI 40 FUEL/CLOSE-OUT REPORT
ARR 04 ARRIVAL
AVR 19 AIR CREW ORIGINATED VOICE REQUEST
CHO 15 CHANGEOVER OR IN-RANGE REPORT
CLK 20 AIRBORNE GMT CLOCK RESET
CLR 08 FLIGHT CLEARANCE
CMD 26 ACARS AVIONICS MEMORY LOAD OR DIAGNOSTIC FUNCTION
CNL 13 CANCELLATION OF FLIGHT
CPL 10 CURRENT FLIGHT PLAN
DEP 03 DEPARTURE
DFD 35 ACARS DIGITAL FLIGHT DATA ACQUISITION UNIT
DIV 41 AIRCRAFT DIVERSION MESSAGE
DLA 12 FLIGHT DELAY
ENG 38 AIRCRAFT ENGINE DATA
ETA 17 ESTIMATED TIME OF ARRIVAL
ETR 32 AIRCREW INITIATED REVISION TO PREVIOUSLY ADVISED ESTIMATED TIME OF ARRIVAL

FAM 14 FLIGHT MOVEMENT ADVISORY MESSAGE
FML 33 ACARS FLIGHT MANAGEMENT COMPUTER - LEFT
FMR 34 ACARS FLIGHT MANAGEMENT COMPUTER - RIGHT
FPL 09 FILED FLIGHT PLAN
FPR 24 AIRCRAFT ORIGINATED REQUEST FOR FLIGHT PLAN UPDATE VIA ACARS
FPU 23 GROUND ORIGINATED FLIGHT PLAN UPDATE TO AIRCRAFT VIA ACARS
GVR 18 GROUND ORIGINATED VOICE REQUEST
HJK 39 AIRCRAFT HIJACKED
LIF 22 GROUND ORIGINATED AIRCRAFT LOAD INFORMATION
MVT 16 FLIGHT MOVEMENT
OAT 21 AIRBORNE OPTIONAL AUXILIARY DATA TERMINAL/DEVICE
PDM 42 POSSIBLE DUPLICATE MESSAGE
POS 02 POSITION WITHOUT WEATHER
PSN 10 AIRCREW INITIATED POSITION REPORT WITH/WITHOUT WEATHER INFORMATION
RDO 25 ACARS AVIONICS MEMORY READOUT
RTN 29 ACARS EQUIPPED AIRCRAFT RETURN TO GATE
SPL 11 SUPPLEMENTAL FLIGHT PLAN
SVC 37 ACARS COMMUNICATIONS SERVICE MESSAGE
THR 31 AIR CREW INITIATED OR AUTO SENSED TAKE OFF THRUST
TIS 06 AIRPORT TERMINAL INFORMATION SERVICE
WXO 28 WEATHER OBSERVATION REPORT
WXR 27 WEATHER OBSERVATION REQUEST

Output

The *OMP* does not display any windows or produce log files. The *OMP* responds to the ETMS *Net.mail* S0 command and provides a statistical report.

TO buffers

The *OMP* ships the buffers of TO messages to all specified applications when either the buffer ages too much or the buffer becomes full.

Statistical Reports

The *OMP* formats and ships a status report to users who request them via the *Net.mail* S0 command.

Processing

The *OMP* registers to the *ARINC* process of the ETMS to receive all messages read from the *ARINC* network. The *OMP* then examines each message. If the message is a position report, the message is parsed. The parsed message is then converted into an ETMS TO position report, buffered and when either the buffer ages sufficiently or becomes full, the *OMP* process forwards the buffer to specified applications. The *OMP* reads from its configuration file, at initialization, a list of ETMS application addresses and forwards each buffer to all addresses. The *OMP* is composed of one processing module, DOTSTO.

The *OMP* is composed of three phases: initialization, processing, and termination.

Initialization. The initialization phase is concerned with reading the three configuration files and configuring the *OMP* process. This includes loading the fix table and the token table into memory, connecting to the ETMS message switching system (node.sw), and registering to the *ARINC* process for the position reports.

Processing. The processing phase is one large loop. Refer to the PROCESS procedure below for further details. The processing phase maintains the registration to the *ARINC* process, reads *ARINC* position reports, translates the reports into TO messages, buffers the messages, and ships the buffers to all specified recipients.

Termination. The termination phase is concerned with releasing all resources. This includes closing all files, unregistering to the *ARINC* process, closing the ETMS message switching connection, and terminating in a safe way to facilitate the next invocation of the process.

Error Conditions and Handling

The *OMP* maintains a trace log. Various exceptional events are written to this file. The file is constrained to 65K bytes. The file is cleared and written again whenever 65K bytes are in the trace file.

The *OMP* treats the following syndromes as fatal during initialization:

- There is no ETMS path structure (pgm_get_etms_path failed). This implies that the directory structure is unknown.
- The first argument (adaptation file) is missing.

The following errors are handled transparently:

- inability to connect to node.sw
- configuration file missing
- fix file missing
- token file missing
- loss of *ARINC* data (registration loss, disconnect from node.sw etc.)

A standard ETMS cleanup handler within the *OMP* server allows the *OMP* to terminate gracefully while carefully releasing all its owned resources.

7.5.1 OMP Routines and Procedures

atoi

Atoi converts an ASCII text string into an integer. The sign is maintained.

buffer_add

Buffer_add places the completed TO message at the end of the buffer. If the buffer overflows, send_to_nas is called to send the filled buffer. If the TO is over 1024 bytes, it is considered to be an error and is ignored. Otherwise the TO is placed onto the buffer.

buffer_init

This routine formats the buffer header by adding the security field followed by an SQ message. The sequence number and the timestamp for the SQ message are also computed. This procedure also sets the time at which this buffer would age and have to be sent.

calculate_speed

This procedure computes the speed based upon two position points and the time at the specified positions. The speed determination is made by calculating the great_circle distance between the two points and dividing this by the delta of the times. If the times are not known or the distance calculation fails, 0 is returned for the speed.

check_sw_mbox

This procedure polls the *node.sw* connection for any messages and dispatches any received messages to the appropriate processing routine. The event counters for the connection are updated accordingly. This routine performs an internal read cycle until there are no more messages.

The network addressing warning message of WARN_NET_T_GET_NEW_EC causes a call to get_new_ecs.

If there is a fatal read error, a sequence of calls to close_sw_mbox and open_sw_mbox is made prior to returning.

Otherwise the type of message is determined and the appropriate processing routine is called:

returned message -> process_returned_message
arinc_msg_data -> process_arinc_message (also reset no_data timer)
net_t_reconfigure -> process_reconfigure
NET_T_GIVE_STATUS_LEV_0 -> process_stats_lev_0
all other stats requests -> stats_not_supported

All other messages cause an error message to be sent to them stating that it is an unknown message type to DOTS.

cleanup_handler

This procedure performs an orderly shutdown by calling close_sw_mbox and then closing the error trace file.

clear_flight_record

This routine zeroes out all entries in the default template record. As each field of an *ARINC* is parsed, it is placed into this default template record.

close_sw_mbox

This procedure closes the connection to the ETMS message switching system (node.sw).

create_to

This routine formats the TO message by using the fields as filled in within the default template record. This routine also calls calculate_speed to do as its name implies.

This routine also maintains a list of the last ten TO messages generated. This list is included within the response to the S) *Net.mail* command.

enter_fix

This routine creates an entry on the linked list for the specified fix name. The fix name is first hashed by *hashit*, and the result is used to determine the linked list location to place the entry. If there is no entry at the linked list for the entry, it is created. Otherwise, the linked list is traversed until the end is reached, and the entry is then appended onto the list.

extract_number

This routine is part of the logic that breaks up an word from the adaptation file into an ETMS address. This routine extracts the value from the address; it is usually separated by a period or close parenthesis. This routine then returns the binary value for the ASCII specified. A zero is returned if the format is invalid.

extract_three_words

This routine is part of the initialization logic. It returns three words from an input line (from the adaptation file). The routine get_word is called first. If the word is not two to three characters long or begins with a #, the routine returns immediately. Extract_three_words then calls get_word again. If the second word is not exactly two digits, this routine terminates. The numeric value is then computed. The remainder of the line is then extracted, leading nondigit or nonletters are dropped, and trailing spaces are dropped. The surviving characters, up to eighty long, are then returned.

fix_lookup

This routine looks up the specified fix in the fix linked list. The resultant latitude and longitude is then returned. The lookup is done by first hashing the fix name (by calling *hashit*) and then traversing the indicated linked list for an exact match. If the fix name is not found, this routine returns false.

get_day_of_month

This routine returns the day of the month based upon a time input.

get_new_ecs

This routine returns new event count values for network addressing. This routine is called whenever the API system indicates that there is a need to obtain new event counters.

get_token

This routine extracts the next token from the *ARINC* message buffer. It first skips over any characters that are not upper case letters, digits, or the * character. This routine then moves the characters of upper case letters, digits, or the * character. Early termination occurs if the end of the buffer is found or eigthy characters have been moved.

hashit

This procedure takes an ASCII word and performs a tailored arithmetic hash operation on it to generate a pseudo random address. The resultant address is then used to indicate which linked list header to traverse to add or extract a fix entry.

initialize

This routine configures the *OMP* process so that it is able to perform its functions. The following operations are performed:

- Connect to ETMS message switching by calling open_sw_mbox.
- Setup start time.
- Determine the ETMS file structure (i.e. /etms5) and upon failure terminate.
- Create the necessary working environment (create file: </etms5>/dots/trace/dots.error and the </etms5>/dots/data files).
- Setup the time event counter.
- Obtain the configuration file (argument 1) and if none is found, terminate.
- Invoke the function read_adaptation_file to do its namesake.
- Create the memory mapped fix file by calling map_fix_rec and then fill in the memory mapped file by calling read_fixfile to read the fixes into memory from file (etmss)/DOTS/FIX.DAT
- Read in the SMI token file by calling read_file.

is_eol

This routine checks to see if the supplied character is an end of line character (ASCII NL)

latlong_to_short

This routine translates an ASCII lat/lon value in the twelve-byte format of ddmmA/dddmmA into two binary values. If A is S or E, a negative value is returned. Otherwise the returned value is either dd*60+mm (for latitude) or (ddd*60+mm) for longitude.

map_fix_rec

This routine creates a memory mapped file that will contain the fix names and their respective latitude and longitude values.

match_smi

This procedure performs a binary search on the SMI table looking for the keyword. If the keyword does not exist, a -1 is returned, otherwise the table index is returned.

open_sw_mbox

This routine formats the call to the API toolkit of net_open to connect to the node.sw. The connection class is DOTSTO. On a successful connection, the event counters are appropriately primed. On a failure, the event counters are set to a one-minute timer to try again.

parse_message

This routine controls the parsing of *ARINC* messages into ETMS TO messages.

Parse_message begins by calling clear_flight_record which initializes the global workspace that will contain fields as they are translated.

If the first word is PDM (possible duplicate message), the buffer is moved to lose the indicator.

If the first four characters are POS (the message is a POS message), invoke the process_pos routine, and advance to the finish_up label.

If the first four characters are (RCL), the message is a RCL message; invoke the process_rcl routine, and advance to the finish_up label.

If the message was not one of the above cases, extract the first word in the buffer. If the word is not FI, it must be three characters, otherwise the message is ignored. If the three characters are not in the SMI table (calling match_smi to look at the table read in at initialization), the message is ignored.

To complete and summarize, if the first word was not FI and the first word was in the SMI table, get the next word by calling get_token.

A non (POS) and a non (RCL) is then processed by repetitively checking the value of the most recently read token and calling the appropriate processing routine. This checking loop completes when the processing routine returns an error, there are no more tokens, or the processing routine signals that enough data has been extracted from the message to make a complete TO message.

The following table is then used:

token	processing routine
FI	process_fi
NP	process_np
OV	process_ov
EO	process_eo
AL	process_al
WV	process_wv
DS	process_ds
DA	process_da
OF	process_of

A token of DT marks the end of the processing for this message.

After each token is processed, get_token is called to perform another iteration. The processing terminates when there are no more tokens.

The finish_up label is utilized after all the above processing completes. This label signals the validation of the now populated global flight record. If there is not at least one position and one time, the message is ignored.

This procedure completes processing by calling create_TO and buffer_add to translate the flight record into a TO message and to buffer it for sending to the specified clients when either the buffer fills or times out.

process

This routine is in essence an infinite loop. It calls check_sw_mbox to get messages from the ETMS, if the timer goes off, it calls send_to_nas to send any buffered data and it then waits for an event counter. If the connection to node.sw is invalid, it will also attempt to reconnect by calling open_sw_mbox on a timed basis.

process_al

This routine processes the AL (altitude) token. Altitude may have a prefix of C, D, or L and be five or 9 characters long. If this is so, strip off first character before processing the ASCII digits.

The altitude must be four or eight characters and the first character must be A or F. The next three characters must be digits. These three digits are the altitude and they are copied into alt1, alt2, and alt3 fields of the global flight record.

Any errors cause the flight record to be marked invalid.

process_arinc_message

This routine takes a quick look at all messages from *Arinc*. If the message does not begin with the QU priority field, it is ignored. Otherwise, the skip_header routine is called to remove the *ARINC* header and the main parsing routine parse_message is activated to do its job.

process_da

This routine processes the DA (departure aerodrome) token. This routine begins by calling the get_token routine. This token is the departure airport, which is checked for a length of three or four characters. The valid airport name is then moved into the dept field of the global flight record.

Any errors cause the flight record to be marked invalid.

process_ds

This routine processes the DS (destination aerodrome) token. This routine begins by calling the get_token routine. This token is the destination airport, which is checked for a length of three or four characters. The valid airport name is then moved into the dest field of the global flight record.

There is allowance for an optional ETA field. The buffer position is saved and get_token is called. If the token is not four digits, the buffer position is restored and this routine completes its processing.

This possible ETA field is validated for hours being the 00-23 range and for minutes being in the 00-59 range. If this is so, the ETA is moved into the global flight record dest_eta field.

Any errors cause the flight record to be marked invalid.

process_eo

This routine handles the EO (estimated to be over a position at a time) token. It begins by calling process_lat_lon_time. When process_lat_lon_time is compeleted, the returned values are moved into pos2 and eta2 of the global flight record.

process_error

This routine logs error messages into the error_stream (and erases the error stream if it is too large) and sends an error message to the specified ETMS address.

process_fi

This routine processes the first field in most position reports, the aircraft identifier. It must be two to seven characters with the first characters being an upper case letter and must contain all upper case letters and digits.

This routine calls get_token to obtain the aircraft identifier and if it does not satisfy the above rules, the global flight record is marked as being invalid.

A valid aircraft identifier is moved into the acid field of the global flight record.

process_lat_lon_time

This routine extracts the position and time at the position from a buffered *ARINC* message. This routine begins by calling get_token.

A fix name is two to six characters with the first character being an upper case letter. If the fix name is valid, based upon a call to fix_lookup, advance to the OBTAIN_ETA label. There is a special case for SN???W where the W is removed prior to the checking of the fixes.

If the above rule did not work, check for altitude (refer to process_al for further details) which is either A or F followed by three digits. If it is an altitude, reset the buffer pointer before the get_token in this routine, and return. This is an altitude and not a position field.

If it is not an obvious fix or altitude, check for an ETA, save the buffer position, and call get_token. If the token is exactly four digits, return; otherwise reset the buffer pointer before this possible ETA and return.

If it is not an easy fix, altitude or ETA, try to work this as a lat/lon field.

This routine counts the consecutive digits. If there are not one to four digits, this is not a latitude; advance to the OBTAIN_ETA label. Otherwise move in the latitude and pad it out with trailing zeroes until there are four digits moved.

If there are no more characters, get the next token.

Move on the subsequent N or S, if neither move in N. The latitude is now complete move in the / character and look at longitude.

If there are no more characters, get the next token.

This routine counts the consecutive digits. If there are not one to five digits, this is not a longitude; advance to the OBTAIN_ETA label. Otherwise move in the latitude and pad it out with trailing zeroes until there are five digits moved.

If there are no more characters, get the next token.

Move on the subsequent E or W, if neither move in W.

This routine then validates that the latitude/longitude is valid; otherwise it clears the field.

The label OBTAIN_ETA saves the buffer pointer and then calls get_token. There must be four digits such that the hours and minutes are in the ranges 00-23 and 00-59, respectively. If the time field is valid, call get_day_of_month to compute the day to apply to this time; otherwise reset the buffer pointer, and return.

process_np

This routine processes the next point by calling process_lat_lon_time. The resulting data is moved into pos3 and eta3 of the global flight record.

process_of

This routine processes the time off (departure time) field. This routine calls get_token to obtain the time field. If the field is not four digits, the buffer pointer is saved, and the next token is obtained and checked. If the second token is not four digits, the buffer is reset and no time is returned.

In either case the four digits are checked to see that the hours and minutes are in the ranges 00-23 and 00-59, respectively. If the time is invalid, it is ignored; otherwise it is copied into field dept_time of the global flight record.

process_ov

This routine processes the time over a position field; it calls process_lat_lon_time. The resultant information is moved into fields pos1 and eta1 of the global flight record.

The next field may be an optional altitude. Save the buffer pointer and call get_token. If the first character is A or F followed by three digits, it is an altitude field. This routine moves the altitude into fields alt1, alt2, and alt3 of the global flight record. Otherwise the buffer pointer is reset to its saved position.

process_pos

This is a fixed field position report. This routine calls get_token twice. The aircraft identifier (second token) is then validated. If it is not two to seven characters with the first character being an upper case letter and all other characters being upper case letters and digits, the global flight record is marked as being invalid.

After the aircraft identifier is moved into the global flight record, process_lat_lon_time is called and the flight record fields pos1 and eta1 updated accordingly. This is repeated again

with fields pos2 and eta2 being updated. This routine calls get_token a third time for lat/lon information. If there is no third position, this routine completes processing. Otherwise process_lat_lon_times is called a third time, and pos3 and eta3 of the global flight record are updated.

process_rcl

Refer to process_pos for details.

process_reconfigure

This routine re-reads the configuration file upon user reconfigure command.

process_return_message

This routine logs all returned messages into the error file. The check_truncate file routine is called to keep this file from growing too large.

process_stats_lev_0

This routine processes a user S0 command from *Net.mail*. It returns to the user a formatted report.

process_wv

This procedure skips over the WV fields. This is done by calling get_token, saving the buffer position, and then calling get_token again. If the second token is AT, get another token; otherwise reset the buffer position. The ETMS does not use the WV fields.

put_data

This routine performs generic transmission for the *OMP* module. It sends messages to the ETMS messages switching system.

read_adaptation_file

This routine reads in the *OMP* adaptation file. All empty lines and lines with '#' as the first printing character are ignored. Processing is terminated on an end of file condition.

All entries in the file are presumed to be addresses or the word *new*. The word *new* sets a flag that is not currently in use.

The address must be in the format (xx.yy.zz). Where xx is site id, yy is node id and zz is the class identifer.

The following cycle is performed until the end of the file or the address table is filled:

```
get_line
get_word
extract_number                    -                    site_id
extract_number                    -                    node_id
extract_number                    -                    class
validate site by calling net$_inq_site_ascii
```

The remainder of the list is ignored when the maximum entries is reached (MAX_SITES).

read_file

This procedure reads in the key_word table (SEI codes). The list is in alphabetical order, and each line is processed by a call to extract_three_words.

read_fixfile

This procedure reads in the list of fixes. The format is a fix name followed by latitude and longitude. The enter_fix routine does the work of translating the lat/lon values and loading up the table.

send_to_nas

This routine sends the buffer when either it fills or times out. If there are no clients, buffer_init is called; otherwise this routine does nothing except re-arm the timer. Otherwise the buffer is shipped out in groups of eight addresses. The buffer is then cleared by calling buffer_init.

skip_header

This routine removes the header by looking for the STX character. The buffer is then moved up to overwrite the header.

stats_not_supported

This routine returns to the requestor a short message indicating that the requested statistics report is not supported by this program.

time_to_short

This routine converts an hour and minute field to a 16-bit integer.

7.5.2 Source Code Organization

The *OMP* consists of one source code module (dots_red.c) and one insert file (dots.h).

Building Instructions

The *OMP* is built by using the following UNIX Makefile:

```
#*********************************************************
#*********************************************************
#                    ETMS          VERSION          5          OPEN          SYSTEMS
#                                                  DOTS                      MODULE
#*********************************************************
#*********************************************************

#---------------------------------------------------
#              Directories              for              this              specific              software
#---------------------------------------------------
ROOT_DIR =                                                                          /RLM.osc
SHARED_DIR                                  =                                $(ROOT_DIR)/shared
API_ROOT_DIR                                                        =          $(ROOT_DIR)/api
DOTS_DIR  = .

#---------------------------------------------------
#              Where              to              find              PasANSI              stuff
#---------------------------------------------------
PASANSI_SRC_DIR      =                                                  $(ROOT_DIR)/PasANSI
PASANSI_INC_DIR      = $(ROOT_DIR)/PasANSI

#---------------------------------------------------
#              Where              to              find              API              stuff
#---------------------------------------------------
API_SRC_DIR                         =                        $(API_ROOT_DIR)/sources/api_openlib
API_INC_DIR                         =                                        $(API_SRC_DIR)
API_LIB_DIR                         =                                        $(API_SRC_DIR)
API_LIB                             =                                                opensys
API_LIB_FULLNAME    = $(API_LIB_DIR)/lib$(API_LIB).a

#---------------------------------------------------
#              Compiler              flags              and              API              lib
#---------------------------------------------------
CC=cc
#CFLAGS    =    -g    -I$(DOTS_DIR)    -I$(API_INC_DIR)    -I$(PASANSI_INC_DIR)    -I$(SHARED_DIR)
CFLAGS      = -0 -I$(DOTS_DIR) -I$(SHARED_DIR) -I$(API_INC_DIR) -I$(PASANSI_INC_DIR)

#-----------------------------------------------
#                                                                                Libraries
```

```
#-------------------------------------------
LIBS = -L$(API_LIB_DIR) -lopensys -lm

#-------------------------------------------
#                    Library                    dependencies
#-------------------------------------------
API_INCLUDES                          =                    \
   $(API_INC_DIR)/api_calendar.h\
   $(API_INC_DIR)/api_error.h\
   $(API_INC_DIR)/api_evt.h\
   $(API_INC_DIR)/api_file.h\
   $(API_INC_DIR)/api_ios.h\
   $(API_INC_DIR)/api_lib.h\
   $(API_INC_DIR)/api_machine.h\
   $(API_INC_DIR)/api_mbx.h\
   $(API_INC_DIR)/api_misc.h\
   $(API_INC_DIR)/api_mutex.h\
   $(API_INC_DIR)/api_name.h\
   $(API_INC_DIR)/api_net.calls.h\
   $(API_INC_DIR)/api_net.h\
   $(API_INC_DIR)/api_pad.h\
   $(API_INC_DIR)/api_pfm.h\
   $(API_INC_DIR)/api_program.h\
   $(API_INC_DIR)/api_set.h\
   $(API_INC_DIR)/api_shmem.h\
   $(API_INC_DIR)/api_vfmt.h
```

```
API_LIBRARY = $(API_LIB_DIR)/libopensys.a

#-------------------------------------------------------
#                    Dependent                    include                    files
#-------------------------------------------------------
DOTS_INCLUDES                                   =                                   \
        $(API_INC_DIR)/api_lib.h\
        $(SHARED_DIR)/etms.lib.h                                                    \
        $(PASANSI_INC_DIR)/PasANSI.h\
        $(DOTS_DIR)/dots.h

#-------------------------------------------------------
#                    Dependent                    object                    files
#-------------------------------------------------------
OBJECTS_DOTS                    =                    $(DOTS_DIR)/dots_read.o        \
                    $(SHARED_DIR)/etms.lib.o                                        \
                    $(DOTS_DIR)/ftm_great_circle.mathlib.bin                        \
                    PasFile.o

#-------------------------------------------------------
#                      What                      to                      build
#-------------------------------------------------------
all:                                                                        dots
#-------------------------------------------------------
#                    How                    to                    build                    it
#-------------------------------------------------------
dots:                                                                   $(OBJECTS_DOTS)
        @echo                                   '[linking                       $@]'
        $(CC) -o $@ $(OBJECTS_DOTS) $(LIBS)

#-----------------------------------------------
#                  all                  of                  the                  .o                  files
#-----------------------------------------------

dots_read.o:                                    $(DOTS_DIR)/dots_read.c         $(DOTS_INCLUDES)
        @echo                                   '[compiling                     $@]'
        $(CC) $(CFLAGS) -c  $(DOTS_DIR)/dots_read.c

PasFile.o:                                      $(PASANSI_INC_DIR)/PasFile.c    $(PASFILE_INCLUDES)
        @echo                                   '[compiling                     $@]'
        $(CC) $(CFLAGS) -c  $(PASANSI_INC_DIR)/PasFile.c

$(SHARED_DIR)/etms.lib.o:
        cd $(SHARED_DIR); make etms.lib
```

OMP Constants

```
#define NET_NIL (-1)        /* VALUE OF HANDLE ON FAILED OPEN AND ID OF FAILED INQUIRE */
#define                     TIME_EC                                             1
#define                     GET_EC                                              2
#define                     PUT_EC                                              3
#define                     MAX_EC                                              3
```

```
#define MAX_SITES (NET_MAX_ALLOWED_ADDRESSES*2)
#define MAX_ETMS_SEQUENCE 9999
#define MAX_BUFFER_SIZE 6000
#define LINE_SIZE 512
#define BUFFER_SIZE 512
#define THIRTY_SECONDS (30*4) /* THIRTY SECONDS USED FOR EVENT COUNTER ARMING */
#define THREE_MINUTES (3*60*4) /* THREE MINUTES USED FOR EVENT COUNTER ARMING */
#define TWENTY_MINUTES (20*60*4) /* TWENTY MINUTES USED FOR EVENT COUNTER ARMING */
#define BUFFER_TIMEOUT_SECONDS 30
#define FIX_REC_SIZE (sizeof(FIX_REC))
#define MAX_FIX_ENTRIES 5000
#define ARINC_MSG_DATA 16435
#define DOTS_SECURITY_CODE " "
#define fix_file "fix.dat"
#define smi_file "dots_smi"
#define NAME_LEN 10
```

7.5.3 OMP Data Structures

```
typedef struct {
    char acid [7];
    char pos1 [12];
    char pos2 [12];
    char pos3 [12];
    char alt1 [4];
    char alt2 [4];
    char alt3 [4];
    char eta1 [7];
    char eta2 [7];
    char eta3 [7];
    char dept [4];
    char dest [4];
    char dest_eta [4];
    char dept_time [4];
    int valid;
    short speed;
  } FLIGHT_REC;

typedef struct FIX_REC {
    char name[NAME_LEN];
    char lat[NAME_LEN];
    char lon[NAME_LEN];
    char magv[NAME_LEN];
    char dsply;
    struct FIX_REC * next_rec;
  } FIX_REC;

typedef FIX_REC * FIX_REC_PTR;
typedef char char3_t [3];
typedef char char8_t[8];
```

```
typedef                 char                                    char10_t[10];
typedef                 char                                    char12_t[12];
typedef                 char                                    char32_t[32];
typedef                 char                                    char80_t[80];
typedef                 char                                    char256_t[256];
typedef                 char                                    char512_t[512];
typedef                 char                                    char1024_t[1024];
typedef char    char8192_t[8192];

typedef                         char3_t                 KEY_ARRAY       [256];
typedef                 int                             KEY_INDEX       [256];
typedef  char80_t  KEY_TEXT [256];
```

7.6 ARINC and NADIN Servers

Purpose

Arinc.*Server* is a dual program. It is either *Arinc.Server* or *Nadin.Server*. A compilation switch determines which server type it becomes. The code is virtually identical; the exceptions are address length, priority field format, message length, and message framing. The differences will be identified where needed in this section. Unless stated otherwise, the word *Arinc* can be used to mean *Arinc* or *Nadin* in this section.

The *Arinc*.*server* is described in this section, and it intimately interacts with the *Arinc*.driver. For simplicity, *Arinc*.*server* and *Nadin*.*server* are called *Server*; *Arinc.driver* and *Nadin.driver* are called *Driver*.

Input

Runtime Parameters. There is only one argument to the *Server* and it is optional, the name of the configuration file. If the argument is missing or invalid (it must begin with a /), the following defaults are used:

Arinc: /<etms5>/arinc/config/arinc.server.config
Nadin: /<etms5>/nadin/config/nadin.server.config

Statistical Requests. See *Arinc* and *Nadin* Processing.

Configuration File Format. Note that any blank line or any line beginning with # is ignored. The first section (or line) contains the *Server* mailbox name. This must match the name specified in the *Driver* configuration file. A typical line is:

/mbx/arinc.mbx.

The second section contains the class to address mapping table. There can be up to fifty classes (valid lines) with a maximum of five *Arinc* addresses per line. A typical line is:

PRINTR ACYZZYA TFMBSYA

The class table must be terminated with a line beginning with the word END.

The third section is a list of addresses to be notified of various noteworthy events (users to be notified), *Arinc* up/down, messages timed out, messages being rejected. There can be up to 16 lines of addresses with one address per line. The format of the address is:

(aaa.bbb.ccc), where aaa is site, bbb is node, and ccc is class

For example, (6.4095.9) causes errors to be sent to all occurrences of class nine on site six (all *Net.mail* on $vntsca).

The notification address list must be terminated with a line beginning with the word END.

The fourth and final section is the auto distribute list. This is a list of addresses and the class of messages being received that will be sent to the address. Up to 16 addresses can be specified. One address and one class are allowed on each line. The receiver of these messages must acknowledge them, or they will be resent. The format of each line is (aaa.bbb.ccc) DDDDDD, where aaa is site_id, bbb is node_id, ccc is the class field of the network address, and DDDDDD is one of the classes specified in the second section of this configuration file.

For example, (6.4095.9) SI would cause each SI type message to be sent to (6.4095.9) until either an acknowledgment is received or the message times out.

Output

Test Message Generation. Test messages can be sent to *ARINC* by using the keyword TEST 1 or TEST 2 on a *Net.mail* M command to the *Server*, i.e., M $this.all.nadin.all TEST 1.

The *Driver* can be reset by using the keyword RESET on a *Net.mail* M command, i.e., M $this.all.nadin.all RESET.

Hourly Archives. All messages to or from *Arinc*, with appropriate control information, are written into files which are created for each hour, the transmit files are called XMT, and the receive trace files are called MSGS.

All messages from the *ARINC* network are written to a log file (/arinc/msgs.date/time), e.g., //jfk/arinc/msgs.960809190031.

All messages to the *ARINC* network are written to a log file (/arinc/xmit.date/time), e.g., //jfk/arinc/xmt.960809190031.

All messages from the *NADIN* network are written to a log file (/nadin/msgs.date/time), e.g., //jfk/nadin/msgs.960809190031.

All messages to the *NADIN* network are written to a log file (/nadin/xmit.date/time), e.g., //jfk/nadin/xmt.960809190031.

Processing

Server is the front end for the *ARINC* driver. It receives message blocks from the *Driver* and passes the messages to the appropriate clients. It also forwards messages from clients and/or com_server mailboxes to the *ARINC* network.

Statistical Requests. The *Server* program responds to the following statistics requests from *Net.mail.*

A *Net.mail* **S0** request returns the large statistics list. This **S0** report lists the *Driver* connection, its state, message counts, and their timing.

A *Net.mail* **S1** request causes a poll to be sent to the *Driver*. The information received from the *Driver* is reformatted by *Arinc.server* into a simple table.

A *Net.mail* **S2** request returns the clients registered to *Server* and the adaptation table of addresses to client connection types.

A *Net.mail* **S3** request returns the list of messages queued to the *Driver*.

Receiving Arinc Data from the Arinc Server. There are two ways to receive data from the *Server*: *register* and *auto distribute*.

Register. A program registers to *Server* to receive a class of data. This is a standard register_to_service_provider interchange. The class of services is specified in the client connection types as read from the *Arinc* configuration file.

Each message from *Arinc* has its destination address checked by *Server*. If the address is listed for a class, the registered table is checked. If there is a process registered for that class, the message is sent to the registered user's address.

Auto Distribute. *Server* maintains an auto distribute list as read from its configuration file. This is a list of classes and the receiver's address. The node_id field of the address is typically wildcarded to all. The class must be present in the *Arinc* configuration file to be valid.

As with registered users, the destination *Arinc* address is checked against the class table. If there are any matches, the auto distribute table is checked. If the class is present, a message is sent to each version 5 address that was read from the auto distribute section of the *Arinc* configuration file. There is, however, a protocol used between *Server* and an auto distribute destination address.

For auto distribute messages that are explicitly stated in the adaptation file, a message code of CT_OUTPUT_MESSAGE_CODE is sent. The first 4 bytes of data are a sequence number. If *Server* does not receive a message code of CT_OUTPUT_MESSAGE_ACCEPT_CODE with the same four bytes of data, it will resend the message at periodic intervals. If the message is never acknowledged by the receiver it is eventually discarded.

Sending Messages To Arinc Via Server. There are two ways to send messages to *Arinc* using *Server*:

- *mailbox*
- version 5 techniques

Mailbox Sending. The mailbox method is archaic and should not be used. A client process connects to *Server* as a valid client class. *Server* checks every subsequent message from the client process for valid format. If the message is in a valid format, it is queued for shipment to the *Driver*.

Version 5 Sending. The version 5 technique for sending a message to *Arinc* requires the message code to be CT_OUTPUT_MESSAGE_CODE (16435), the first four bytes of the message to be a sequence number, and the following three bytes to be QUu for *Arinc* or FFu for *Nadin*. *Server* will then echo the message back to the user with a message code of CT_OUTPUT_MESSAGE_ACCEPT_CODE (16436).

Handling messages to Arinc. Every message being output to *Arinc* is queued into a permanent file that is reused. This allows for an *Arinc* message that was queued by *Server* to survive the termination and restart of the *Server* and still be sent to *Arinc*. The message is removed from the queue under three circumstances:

- It was successfully transmitted.
- It timed out.
- The queue overflowed. (This very rarely occurs.)

The last two cases cause the message to be returned to sender, an error message sent to the logger process, and an entry made into the current hour's XMT file.

The processing of messages to *Arinc* is a singly threaded pipeline. All messages are queued by the *Server*. A message is then sent to the *Driver* from the *Server*. The *Driver* must successfully transfer the message to the *Arinc* network before the *Server* will remove the message from the queue and give the subsequent message to the *Driver*.

Every message within the *Server*, when it is added to or retrieved from the queue, is validated for format. There are three checks made:

- Excessive line size

- Excessive message size
- Excessive nonprinting characters within the message (Carriage return and line feed are presumed printable in this context.)

If the message fails these checks, it is deleted from the queue, logged into this hour's XMT file, and discarded.

7.6.1 ARINC and NADIN Routines and Procedures

Procedure *add_to_arinc_queue* takes messages from other processes and buffers them into a secure area until they can be sent to the *Driver*. This procedure begins by calling validate_arinc_format and if the message fails validation, returns an error flag to its caller.

This procedure formats a queue entry of queue_rec_t, updates the header record of the queue file, and then writes the entry into the permanent queue file.

If the queue is filled, *Server* deletes the oldest entry, notifies logger and all users to be notified (section three of the configuration file), and makes an entry into this hour's XMT file.

Procedure *auto_distribute_ack_received* matches acknowledgments from auto distribute users to the acknowledged message and receives the message from the nonpermanent auto distribute buffer. The source address is the primary key on the match with the sequence number being the secondary match key.

Procedure *auto_distribute_try_again* resends messages from the nonpermanent auto distribute buffer. There is an enforced time interval between sending messages to the same address. When the interval is exceeded, the appropriate message is resent. If the message times out on this auto distribute buffer, the message is removed and a notification is logged to the xmt hourly files.

Procedure *auto_distribute_add_to_q* attaches an auto distribute message to the end of the specified auto distribute linked list. This procedure sets the approriate timeout and control information for the entry.

Procedure *auto_distribute_flush_q* destroys the auto distribute linked lists. This is used on a reconfigure or program termination.

Procedure *check_q_for_timeout* determines whether messages in the *Arinc* to *Driver* queue have timed out or not. If an entry has timed out, it is deleted from the queue (permanent queue file).

A deleted entry causes the first record in the permanent queue file to be rewritten and the timed out entry to be marked as deleted. All users to be notified are notified, and the entry is logged into this hour's XMT file.

Procedure *check_svr_mbox* reads messages from its *Server* mailbox. It continues reading and processing until there are no more messages. All open requests are sent to mbox_process_open, all closes are sent to mbox_process_eof and all data messages are sent to svr_process.

Each channel is individually checked, and if it exceeds the system limit of bytes transferred, it is closed (otherwise the channel becomes unusable).

Procedure *check_timer* performs scheduled operations. If the auto distribute threshold is exceeded, a call is made to auto_distribute_try_again. If the queue retry threshold is exceeded, a call is made to check_q_for_timeout. If the primary event counter is not satisfied, the routine
does nothing else.

Check_timer then rearms the time event counter. If the NOOP timer is exceeded, a NOOP message is sent to all *Server* mailbox clients.

Procedure *check_truncate_file* is a common utility routine that checks the trace file's current size, and if it is more than 65KB, the file is cleared.

Procedure *clear_chans* deactivates all *Server* mailbox client channels. To do this it sends a close message to each client, waits 0.10 seconds, and then deallocates each channel. This is called when there is a reconfigure or program termination.

Procedure *clear_create_arinc_queues* is called at initialization to recreate the *Arinc* queue. This is a permanent file, so this is called only when the control information in the queue file is corrupted. The queue files are /arinc/queue_file for *Arinc* and /nadin/queue_file for *Nadin*.

Procedure *compare_strings* is a common utility routine that performs an arithmetic comparison of two strings. It does a byte-by-byte comparison until all bytes match or a mismatch is found. A zero is returned if they match, a **-1** is returned if any byte of the first string has a value less than the byte in the same position of the second string, a **-2** is returned if any byte in the first string has a value greater than the byte in the same position in the second string.

Procedure *create_arinc_queue* attempts to load the permanent *Arinc* queue file at initialization. The control portion of the file is validated. If the control portion is invalid or there is any error opening the queue file, create_arinc_queue calls clear_create_arinc_queues to recreate the file. The queue file is retained from one invocation of *Server* to the next. The queue files are /arinc/queue_file for *Arinc* and /nadin/queue_file for *Nadin*.

Procedure *create_arinc_window* is called to make a window when *Arinc* wants to display the fact that there was no data for a number of minutes.

Procedure *create_log_window* creates the two hourly transaction files. All messages to or from *Arinc*, with appropriate control information, are written into files which are created for each hour. The transmit files are called XMT, and the receive trace files are called MSGS.

- All messages from the *ARINC* network are written to a log file (/arinc/msgs.date/time), e.g., //jfk/arinc/msgs.960809190031.
- All messages to the *ARINC* network are written to a log file (/arinc/xmit.date/time), e.g., //jfk/arinc/xmt.960809190031.
- All messages from the *NADIN* network are written to a log file (/nadin/msgs.date/time), e.g., //jfk/nadin/msgs.960809190031.
- All messages to the *NADIN* network are written to a log file (/nadin/xmit.date/time), e.g., //jfk/nadin/xmt.960809190031.

Procedure *create_mbox* creates the *Server* mailbox. The name of the mailbox is read from the *Arinc* configuration file from the first section. *Create_mbox* performs a deletion of any occurrence of the *Server* mailbox file (with the delete when unlocked flag set) to preclude any one program from seizing the mailbox file and not allowing *Server* to control it. The mailbox is then created and the mailbox event counter is obtained via ETMS API calls.

Procedure *display_arinc_msg* is a utility routine that dumps messages from the *Driver* to a specified stream identifier. There is a record structure in use with the format varying by message codes. This routine dumps the information in a consistent format.

Procedure *display_error* writes out error messages to the trace stream and standard output. The trace stream is usually the file /<etms5>/arinc/trace/arinc.server for *Arinc* or /<etms5>/nadin/trace/nadin.server for *Nadin*.

Procedure *display_net_error* writes out version 5 network addressing error messages to the trace stream and standard output. The trace stream is usually the file /<etms5>/arinc/trace/arinc.server for *Arinc* or /<etms5>/nadin/trace/nadin.server for *Nadin*.

Procedure *extract_sequence* is a routine written to extract a record item from a buffer. It is designed to fake out the PASCAL compiler type checking.

Procedure *format_for_driver* is called to format *Arinc* messages requested by a user via *Net.mail*. A message code of net$_msg_data_t with a keyword of TEST 1 causes a short test message to be sent to *Arinc* or *Nadin* via the *Driver*. A message code of net$_msg_data_t with a keyword of TEST 2 causes a longer test message to be sent to *Arinc* or *Nadin* via the *Driver*. A message code of net$_msg_data_t with a keyword of RESTART causes a reset command to be sent to the *Driver*. An **S1** request by the user had been previously translated into an action code of DSTATS, which causes a request for statistics to be sent to the *Driver*. Messages are sent to the *Driver* by calling send_to_driver.

Procedure *get_word_upper_case* extracts a word from the specified buffer. The output buffer is 1024 bytes long, left justified, and blank filled. The returned word is forced to upper case and the length of the word is returned as word_size even though the entire 1024 bytes are blank filled. This routine should be used with caution in that it does not check that the

destination address can contain all 1024 bytes that are blank filled. On some programs this is notorious for destroying other variables or the stack itself.

Procedure *initialize* sets up the *Server* environment. Initialize begins by obtaining the base directory structure by calling the get_etms_path module. The returned path is currently /etms5 and is referred to in this document as /<etms5> to represent that it is externally controlled.

Initialize creates the /sio_files and either the /arinc or /nadin directories. The configuation file is defaulted to either /<etms5>/arinc/config/arinc.server.config or /<etms5>/nadin/config/nadin.server.config. Argument one to the program is then checked. If argument one exists, is over two bytes long, and begins with a /, it overwrites the previous default configuration file name.

Initialize creates the trace file. This is done by creating the /<etms5>/arinc/trace (or /<etms5>/nadin/trace) directory and then creating the trace file. The trace file is constrained to 64Kb long by periodic calls to check_truncate_file within *Server*. The trace file name is either /<etms5>/arinc/trace/arinc.server or /<etms5>/nadin/trace/nadin.server.

Initialize clears the class to address table and the *Server* mailbox client list.

Initialize obtains the time event counter and arms all timers. After performing the time overhead, Initialize calls the following routines to complete the initialization:

- read_adapt_file to load the configuration file
- create_mbox to create the *Server* mailbox
- create_arinc_queue to load the permanent *Arinc* queue
- net_open to interface with the ETMS network addressing system

Procedure *kill_arinc_driver* terminates the *Driver* process. It will not terminate the process until a parameterized interval has been exceeded. It invokes the /com/sigp program to terminate either *Arinc.driver* or *Nadin.driver*. This is typically called when too many messages are timing out in the *Driver*.

Procedure *log_outbound_message* writes received *Driver* control messages into this hour's XMT file. It invokes write_header to apply the standard header format and then writes the approrpriate text for each message type.

Procedure *mbox_process_eof* handles the situation of a close message being received in the *Server* mailbox. This routine performs the table cleanup for the client and deallocates the mailbox channel.

Procedure *mbox_process_open* handles the situation of a user connecting into the *Server* mailbox. The first word of the mailbox message is the desired connection class. This must match a class entry read from section two of the configuration file. If it does not, an open

rejection message is returned to the opening process. If it does match, the second word of the message contains login information, which is retained, and an open accepted message is returned to the opening process. The various parameters for the mailbox channel are then updated. If the connecting class is *ARINC* or *NADIN*, the *Driver* channel and control information are set. If the connecting class is DISPLY, only one is allowed, and the last one in is the valid one. A special flag is set so that all received messages and the allied control information will be sent to the DISPLY channel.

Procedure *move_bytes* is a common utility routine that moves presumed bytes from one string to another. This is another method of *fooling* PASCAL type checking. Any variable type can be the origin or destination string. This routine should be used with caution in that it does not check that the destination address can contain all the bytes that are being moved. On some programs this is notorious for destroying other variables or the stack itself.

Procedure *net_open* connects to the ETMS network addressing system as class *ARINC* or class *NADIN*. The appropriate event counters are obtained and armed.

Procedure *net_poll* reads and processes all messages from the ETMS network addressing system. It runs as a continuous loop until all messages have been read and resolved. If there are no more messages or an error occurs, this routine will return processing to the calling routine.

The event counter for the network is rearmed on each read cycle. If the read returns no messages, this routine returns. If there is a warning error on the read from the network, the event counters are obtained and rearmed again.

If a fatal read error occurs, the error is logged, the network connection is terminated, and a new connection to the network is attempted. In any case, as a result of this error, this routine returns control to its calling routine.

If the received message is a *return to sender*, unregister the user.

All other messages are given to *net_process_message*.

Procedure *process_net_user_command* processes net$_msg_data_t commands from users. If the first word of the command is RESTART or RESET, a restart command is sent to format_for_driver. If the first word is TEST and the second word is **1** or **2**, the send a test message is forwarded to format_for_driver. If format_for_driver is called, a reply is sent to the requesting *Net.mail*. Otherwise nothing is returned.

Procedure *net_process_message* routes messages received from the ETMS network addressing system to the appropriate processing routine. The following table gives the message type and the processing routine:

net$_t_give_status_lev_1	process_s1
net$_t_give_status_lev_2	process_s2
net$_t_give_status_lev_3	process_s3
net$_t_give_status_lev_x	process_s0 (for net.mail commands s4 - s9 ands0)

ct_output_message_accept_code : auto_distribute_ack_received
ct_output_message_code : processed internally - see below
net$_t_reconfigure : processed internally - see below
net$_t_reg_for_services : processed internally - see below
net$_t_msg_data : process_net_user_command (if not destined for driver - see below)

The message type ct_output_message_code signals a message to *Arinc* or *Nadin* with an acknowledgment of receipt back to the sender. The first four bytes of the message are a sequence number. The entire message is given to the sender with a message code of ct_output_message_accept_code. The first four bytes are then removed. The message is ignored if the fifth and sixth bytes are not FF for *Nadin* or QU for *Arinc*.

A message of the type net$_t_msg_data is pre-screened prior to calling process_net_user_command for being a message to the *Driver*. If the first three bytes are QU for *Arinc* or FF for *Nadin*, it is a message to the *Driver* and process_net_user_command is not called.

All messages to the *Driver* are logged into this hour's MSSG file, formatted for the Driver and sent to the *Driver* via the send_to_driver call.

A message of net$_t_reconfigure causes the *Server* to reload all its internal tables. If the password is not valid, the command is ignored, and an error message is returned.

The reconfigure is logged into this hour's XMT file and into the trace file and an acknowledgment is returned to the requesting process. The following procedures are then called to satisfy the reconfigure command:

clear_chans
auto_distribute_flush_q
read_adapt_file
create_mbox

A message of net$_t_reg_for_services contains the number of requested services (class types). If the count is zero then unregister_user is called, otherwise register_user is called. On every received message from the *Driver*, the destination *Arinc* address is checked against the address to class table. Each match causes the registered user table to be checked. The users who are registered for that class will receive the message.

Procedure *process* is the main processing routine of the *Server*. This routine is an infinite loop exited only by program termination. The following operations are performed in this processing loop:

- check to see if it is time to open the next hour's XMT and MSGS files
- call check_timer
- call check_svr_mbox

- call net_poll
- call send_queue
- check timeout on no messages from *Driver*, call process_arinc_rcv_timeout
- check timeout on message sent to *Driver*, call process_arinc_xmt_timeout
- wait for event counter to be satisfied (network addressing message, *Server* mailbox message or time)

Procedure *Process_arinc_rcv_timeout* handles the case of no data being received for a while. If there is no error window currently displayed, it calls create_error_window to make one. Updates are made to the error window once a minute. If a minute has not passed, nothing is done. Otherwise, the window receives an updated *no input* Arinc *data for xx minutes* message, and a copy of the message is sent to all users listed in the adaptation file.

Procedure *Process_arinc_xmt_timeout* handles the case of no acknowledgment from the *Driver* on a message to *Arinc*. There is a safety check that the *Driver* does indeed have a message, and if it does not, nothing is done. Otherwise, an entry is made into the xmt file, a message is sent to all users listed in the adaptation file, and kill_arinc_driver is called to terminate the *Driver*. This covers the case where a driver has problems.

Procedure *Process_driver_stats* handles the statistical response from the *Driver*. Earlier, one or more users had sent an **S1** command to the *Server*. The *Server* then sent a statistical request to the *Driver*. The response is encoded into an ASCII message, framed for ETMS transmission, and sent to all users who are awaiting the **S1** response.

Procedure *Process_driver_status* processes status type messages from the *Driver*. In most cases these are unsolicited responses. The various status messages and the processing that results are:

- ipc_$status_ok signifies that the message was sent to the *Arinc* network. This results in a call to Process_message_sent, and if there are queued messages, a call is also made to send_queue.
- ipc_$status_failed signifies that the message was refused by the *Arinc* network. This results in a call to Process_message_aborted, and if there are queued messages, a call is also made to send_queue.
- ipc_$status_ready signifies that the *Arinc* circuit is now available. All users are notified, and send_queue is called if nothing is outstanding to the *Driver*.
- ipc_$status_alive signifies that the *Driver* is now operational. Users are notified, and send_queue is called if nothing is outstanding to the *Driver*.

- ipc_$status_notready signifies that the *Driver* is in a not ready state. Users are notified by a call to send_to_users.

- ipc_$status_noline signifies that there is no *Arinc* connection. Users are notified by a call to send_to_users.

- ipc_$status_msgrejected signifies that *Arinc* rejected the message. This results in a call to Process_message_aborted, and if there are queued messages, a call is also made to send_queue.

- ipc_$status_noresponse signifies that the *Arinc* network did not send a response to the last message. This results in a call to Process_message_aborted, and if there are queued messages, a call is also made to send_queue.

- ipc_$status_dcpfail signifies that the DCP microcode failed. This means that the *Driver* needs to be reloaded. Users are notified by a call to send_to_users.

- Any other status message is unexpected; it will be logged and otherwise ignored.

Procedure *Process_message_aborted* handles the case where the connected network did not accept the sent message, the format was wrong, the line was down, the line was not turned around, etc. If there is no outstanding message to the *Driver*, there is no processing. Otherwise, the queued message is updated. If the retry count has not been exceeded, this routine does nothing. Otherwise, the message is removed from the queue, users are notified by a send_mail_notice, and the events are logged into the xmt hourly files.

Procedure *Process_message_sent* handles the case of a message being successfully transferred to the *Arinc* network. It removes the message from the queue and logs the event into the hourly xmt file.

Procedure *Process_s0* processes the user request for statistics; the user entered an **s0** command to *Net.mail*. This procedure formats the appropriate statistics and returns the ASCII report to the user.

Procedure *Process_s1* processes the user request for *Driver* statistics, the user entered an **s1** command to *Net.mail*. This procedure adds the user's address onto a pending queue and sends a statistics request to the *Driver*. When the response comes back, it is processed by process_driver_status who formats and returns the ASCII response to the requesting *Net.mail*.

Procedure *Process_s2* processes the user request the message distribution information; the user entered an **s2** command to *Net.mail*. This routine formats the ASCII report of people to receive system notices, the address mapping table, the process registration list, and the auto distribution list. The appropriate statistical information is also recorded.

Procedure *Process_s3* processes the user request for the display of all pending messages to the *Arinc* network. The user entered an **s3** command to *Net.mail*. The message queue is traversed and appropriate information is extracted and formatted into ASCII messages, which are then sent to the requesting *Net.mail* process.

Procedure *read_adapt_file* reads the adaptation file at process initialization or as the result of a user's reconfigure command to *Net.mail*. The appropriate tables and queues are then configured for the *Server* operation.

Procedure *register_user* accepts registrations from users for various services. The services must be listed in the adaptation file class table or else the user will receive a net$_t_msg_reg_bad message; otherwise the user will receive a net$_t_msg_reg_ok message. If the address is already in the service table, the old entry will be replaced. If the service count is zero, the entry is removed from the service table. When the destination address of a message is contained in the requested service, the *Server* sends the message to the registered user.

Procedure *send_buffer* writes a specified message to the mailbox.

Procedure *send_mail_notice* transmits notices that are specific to certain messages to the users on the notification list (obtained from the adaptation file, people who automatically receive notifications). This routine also appends message specific information into the transmitted datagram.

Procedure *send_ok_to_driver* packages an acknowledgment of received messages to the *Driver*. It then calls send_to_driver to forward the acknowledgment.

Procedure *send_queue* sends the first message on the output queue to the *Driver*. If there already is a message outstanding to the *Driver*, nothing on the queue, no *Driver* connected or if the *Arinc* network is down, this routine does nothing.

Each queue entry to be sent is validated for format and age. If the format is not valid, a notice is sent to all interested parties, and the message is destroyed. If the message has expired, the message is logged into the hourly xmt file, a notice broadcast to the broadcast list, and the message is removed from the queue.

A valid message is written to the hourly xmt file as a transmit attempt, and the message is sent to the *Driver*.

If there were no valid entries on the queue, the queue is cleared.

Procedure *send_to_auto_distribute_users* traverses the service (class) table, and if the destination address matched (including wildcards), the auto_distribute table is checked. If the service is present, the auto_distribute address is added to the address list. Whenever the address list fills or all matches are found, the message is sent (by calling send_to_node).

Procedure *send_to_driver* begins by checking to see if the message is the type to be queued (i.e., an outbound message) and if so, calls add_to_arinc_queue to queue it. If there already is a message outstanding to the *Driver* and the message is queued, return. Messages that are not eligible to be queued are control type messages and should be sent immediately.

The message is sent to the *Driver* if the *Driver* is connected and ready.

Procedure *send_to_node* sends messages to the ETMS message switching system. The message is sent via toolkit calls. If there is a fatal error, the connection is reestablished and the message is re-sent. No room in the port/mailbox is not a fatal error.

Procedure *send_to_registered_users* determines what service or class the *Arinc* destination address represents. The list of registered users is then checked for the service. All registered users for this service receive this *Arinc* message.

Procedure *send_to_users* formats notices to the notification list users. The notification list is specified in the adaptation file. This procedure prefaces the user supplied text with a NOTICE: type header and then sends it to the users on the notification list.

Procedure *svr_Process* processes the mailbox that the *Server* created. At this point in time, the only client on this mailbox should be the *Driver*. The following processing is performed on messages from the *Driver* channel of the mailbox:

- Record the time of receipt and clear any *no messages received* flags, and call send_to_users to state that data is again being received.
- If the message from the *Driver* is of type ipc_$stats, call process_driver_stats;
- If the message from the *Driver* is of type ipc_$msg (and it is an *Arinc* or *Nadin* message), send an acknowledgment to the *Driver* by calling send_ok_to_driver.
- If the message from the *Driver* is of type ipc_$status, call process_driver_status.
- Any other message type is displayed, then ignored.

The message type of ipc_$msg is further processed by logging it into the hourly msgs file and if a client is connected to the mailbox as class DISPLAY, the message is sent to the process. The destination *Arinc* address is then extracted and calls are made to send_to_auto_distribute_users and send_to_registered_users. All connections to the mailbox are checked, and if the connection type is the class of the destination address, the message is sent to the approppriate mailbox channel.

Input messages from channels that are not the *Driver* are scanned, and if the prefix matches QU for *Arinc* and FF for *Nadin*, the message is sent to the *Driver* by calling send_to_driver.

Procedure *unregister_all_users* is called on program termination and on a reconfigure command. This routine then sends unregister commands to all registered users.

Procedure *unregister_user* removes a specific user from the registered table. This routine is called when a message is returned to a specified registered user or when a registered user unregisters.

Procedure *validate_arinc_format* checks to see that a message is not too long (3840 characters) and does not contain more than five nonprintable characters (line feed and carriage return do not count).

Procedure *write_header* writes uniform headers into the hourly xmt and msgs files.

7.7 ARINC/NADIN Printer Function

Purpose

The *Arinc/Nadin Printer* function is a printing function that moves received messages from either the *Arinc* or *Nadin* servers onto printers at remote sites.

Processing Overview

The *Arinc* printing function and the *Nadin* printing function use the same source code. There is a runtime switch that determines if the software is supporting *Arinc* or *Nadin* print functions.

The word *Arinc* in this section will refer to common *Arinc* and *Nadin* functions unless the text states that the function is either *Arinc* or *Nadin* specific.

The *Arinc* printing function consists of three modules:

- Sprinter Module - the *Arinc* Spooler *(Arinc.sprinter*)
- Rprinter Module - the remote *Arinc* queue manager (*Arinc.rprinter*)
- Qprinter Module - the *Arinc* queue printer (*Arinc.qprinter*)

Arinc.sprinter receives messages from the *Arinc* function and guarantees delivery to all specified *Arinc.rprinter* modules at remote sites. The *Arinc.rprinter* logs the received messages into hourly files, individual files, and into a shared memory region. The *Arinc.qprinter* polls the shared memory region and places the received messages onto a serial port that is connected to a printer.

7.7.1 Sprinter Module

Purpose

The *Sprinter* module or *Arinc.sprinter* receives messages from the *Arinc* server and guarantees delivery of those messages to specified *Arinc.rprinters*.

Design Issue: Environment

Arinc.sprinter is compliant with ETMS version 5 network addressing and open systems. It has no hardware requirements.

Execution Control

Initialization. The initialization phase creates the output directory, connects to node.sw, loads the adaptation data and creates the queue files. *Arinc.sprinter* queues an initialization message to all specified *Arinc.rprinter* modules.

Termination. The termination phase closes all resources, for example, connection to node.sw and each queue file, and then gracefully exits.

Input

Arguments. *Arinc.sprinter* requires two arguments on its command line. The first argument must be either the word **arinc** or **nadin**. If neither word is specified, *Arinc.sprinter* defaults to being in the 'arinc' mode. The second argument is the name of the adaptation file; if none is specified, the following defaults are used:

</etms5>/arinc/config/arinc.sprinter.config

or

</etms5>/nadin/config/nadin.sprinter.config

Adaptation File. The *Arinc.sprinter* configuration file consists of addresses of processes to receive messages from *Arinc.sprinter*. Each address is in the format of (xxx.yyy.zzz), where xxx is the ETMS site identifier, yyy is the ETMS node identifier (typically this value is 4095 for all nodes) and zzz is the ETMS class number (this is usually 87 for *Arinc.rprinter* and 88 for *Nadin.rprinter*). There is a limit of 16 addresses with one address allowed per line. As is usual, blank lines or lines beginning with # are ignored.

Messages From Arinc/Nadin. The *Arinc.server* must have the address of *Arinc.sprinter* (and *Nadin.server* must have the address of *Nadin.sprinter*) in its configuration file in the auto-configure section. Otherwise no *Arinc* (or *Nadin*) messages will be printed. All messages from *Arinc.server* must have a message code of 16435, and the first four data bytes are presumed to contain the *Arinc* message sequence number. *Arinc.sprinter* sends the same message back to *Arinc.server* with a message code of 16436 as an acknowledgment.

There is no filtering of *Arinc* messages in *Arinc.sprinter*. All messages received from *Arinc.server* are forwarded to all listed *Arinc.rprinters*. The filtering occurs in *Arinc.rprinter*.

Statistics Requests. *Arinc.sprinter* considers S0 and S2-S9 to be the same statistics request and provides a summary of its operation. An s1 command causes *Arinc.sprinter* to send a test message to all listed *Arinc.rprinters* with the address of the S1 requestor.

Output

Log Files. The following are examples of log files:

Arinc.sprinter maintains hourly log files which are named </etms5>/etms_data/arinc/print.yymmddhhmmss or </etms5>/etms_data/nadin/print.yymmddhhmmss. These hourly files contain the original Arinc message and its transition through the queue (i.e. received time, shipment time and acknowledgment time). An example is below:

```
 Sending mcode: 16437 to: (0006.0007.0037.0002)
Address index: 0 Count: 4 Empty: 12 fill: 52
Address index: 1 Count: 3 Empty: 12 fill: 52
Address index: 2 Count: 9 Empty: 96 fill: 52
Address index: 3 Count: 5 Empty: 52 fill: 52
 Sending mcode: 16436 to: (0006.0007.0006.0009)
============= Queueing Sequence: 2482 Printer Seq: 11061
QU ACYZZYA
.OPBWXUA 050807/043320 689/215117
?**KSFO** UNITED AIRLINES - SFO AREA WX BRIEFING ****
SFO TAF05 0759 050759Z 050808 OPBWX TAF: VRB05KT P6SM SCT030
   BKN070
   TEMPO 0810 5SM -SHRA BKN030
   FM1000 15006KT P6SM -SHRA SCT020 BKN040
   TEMPO 1018 2SM RA BKN020
   FM1800 17010KT P6SM -SHRA SCT020 BKN040 PROB40 1803
   VRB15G25KT 2SM TSRA BKN020CB....MIN TEMP 12(54F) MAX
   20(68F)....NEUGEBAUER

UPPER LEVEL LOW WILL MOVE OVER SFO DURING THE DAY WITH RAIN THRU
 THE MORNING AND ISOL TSRA THRU THE AFTERNOON.

  Sending mcode: 16435 to: (0089.0014.0087.0000) (0089.0024.0087.0000) (0090.0014.0087.0000)
(0087.4095.0087.0000)
Address index: 0 Count: 5 Empty: 12 fill: 53
Address index: 1 Count: 4 Empty: 12 fill: 53
Address index: 2 Count: 10 Empty: 96 fill: 53
Address index: 3 Count: 6 Empty: 52 fill: 53
 Received ACK on Seq: 2482 from (0089.0024.0087.0183)
   Ack matched Address (0089.0024.0087), now check the sequence #
   Ack matched Sequence also - all done..
 Received ACK on Seq: 2482 from (0089.0014.0087.0109)
   Ack matched Address (0089.0014.0087), now check the sequence #
   Ack matched Sequence also - all done..
 Received ACK on Seq: 2482 from (0087.0005.0087.0013)
```

```
Ack matched Address (0087.4095.0087), now check the sequence #
Ack matched Sequence also - all done..
Received ACK on Seq: 2482 from (0087.0047.0087.0007)
Ack matched Address (0087.4095.0087), now check the sequence #
ACK Did not match ...
Received ACK on Seq: 2482 from (0090.0014.0087.0298)
Ack matched Address (0090.0014.0087), now check the sequence #
Ack matched Sequence also - all done..
```

Messages. The original *Arinc* message is augmented with a printer sequence number. The *Arinc*.server sequence number (generated by *Arinc.driver*) and the *Arinc.sprinter* sequence number are both sent to all specified *Arinc.sprinter* modules.

Statistical Reports. An S0 and S2-S9 cause the following statistical report to be received:

```
RCVD Data Message From (6.7.38.3.0) To (6.8.9.102.0)
STATS ** NADIN HUBSITE PRINTER QUEUE MANAGER STATISTICS - VERSION 2.07 **
 Elapsed time: 176 18:41:57  Current: 05/06 09:23:53

Adaptation file: /etms5/nadin/config/arinc.sprinter.nadin

 Reconfigured 01/19 22:23:44 by (06.07.09.18)

 Net Open Attempts: 24  Net Open Successes: 15

 Test polls to myself: 44199  Rcvd test polls to myself: 44194
  # times node.sw reopened since no poll reply received 5
  Last poll sent 165.50 seconds ago - No poll in progress
Msgs from NET: 293471  Bytes from NET: 118880608
Messages received from Nadin: 24441
Last NADIN message received at 05/06 09:19:20

 Messages Sent to all recipients: 24444
 Port Filled: 833  Other NET_send errors: 2

 Total acknowledgments rcvd: 98014  No Match: 685
 Total Messages Returned: 126144

[ 0] (87.4095.88.00) Count: 2 Empty: 101 fill: 91
   Msgs rcvd/Queued: 15194 Send Attempts: 131300 Msgs returned: 97415
   Acks Rcd & Matched: 14975  Acks Rcd & Not Matched: 0

[ 1] (89.31.88.00) Count: 2 Empty: 101 fill: 91
   Msgs rcvd/Queued: 15194 Send Attempts: 15265 Msgs returned: 52
   Acks Rcd & Matched: 15176  Acks Rcd & Not Matched: 0

[ 2] (89.32.88.00) Count: 2 Empty: 101 fill: 91
   Msgs rcvd/Queued: 15194 Send Attempts: 15290 Msgs returned: 0
   Acks Rcd & Matched: 15176  Acks Rcd & Not Matched: 0

[ 3] (90.14.88.00) Count: 4 Empty: 91 fill: 91
   Msgs rcvd/Queued: 15194 Send Attempts: 45400 Msgs returned: 28441
   Acks Rcd & Matched: 15067  Acks Rcd & Not Matched: 0
```

Stats s0 and s2-s9 all return this same report. Stats s1 sends a test message to all rprinters. An S1 request results in an acknowledgment that a test message was sent to all *Arinc.rprinters*.

Processing

The processing phase of *Arinc.sprinter* is an infinite loop of receiving *Arinc* messages and shipping them to all specified *Arinc.rprinter* modules. The processing phase then uses timers to retry sending each message until it is acknowledged by the recipient. Refer to the PROCESS routine for more details.

Arinc.sprinter receives unsolicited messages from *Arinc* via the *Arinc.server's* auto-distribute function. *Arinc.sprinter* must acknowledge these messages or *Arinc.server* will continue to re-send each message. *Arinc.sprinter* inserts each message into its queues for each specified *Arinc.rprinter* and transmits the messages to each *Arinc.rprinter*. The *Arinc.rprinter* must acknowledge the message or *Arinc.sprinter* will continue to re-send each message.

Timed Events. *Arinc.sprinter* will attempt to reconnect to node.sw once a minute. *Arinc.sprinter* will close and open up another log file once an hour. *Arinc.sprinter* will resend queued entries every two minutes. *Arinc.sprinter* will poll itself every five minutes (message code is 16437). If the poll is not received back from node.sw within 5 minutes, the connection to node.sw is closed and then reopened. This checks the integrity of the node.sw connection.

Error Conditions and Handling

Arinc.sprinter errors are logged into the hourly log file. Whenever *Arinc.sprinter* starts or stops, it sends a notification to all specified *Arinc.rprinter* modules which is then printed.

7.7.1.1 Sprinter Routines and Procedures

IN_USE_BIT_CLEAR

The queue handler uses a bit mask to mark a record as being in use. This routine clears the in use flag for a particular queue slot.

IN_USE_BIT_SET

The queue handler uses a bit mask to mark a record as being in use. This routine sets the in use flag for a particular queue slot.

IN_USE_BIT_QUERY

The queue handler uses a bit mask to mark a record as being in use. This routine checks the in use flag for a particular queue slot.

READ_ENTRY

As part of the processing to read a queued entry, this routine positions a queue file to a specific location and reads the specified entry.

WRITE_ENTRY

As part of the processing to write an entry to the queue, this routine positions a queue file to a specific location and then writes the specified entry.

DUMP_Q

A diagnostic routine that displays the contents of all queues. This is not used operationally.

DELETE_OLDEST_ENTRY

This routine overwrites the oldest queue entry with a filler record and updates all pointers and counters accordingly.

ADD_TO_ARINC_QUEUE

This routine updates all counters and pointers and ultimately calls write_entry to add an entry onto a specified queue. If the queue is already filled, this routine calls delete_oldest_entry to make room.

ARINC_QUEUE_AND_SEND

This routine takes a received message (or an internally generated notice) and places it onto every queue (by calling add_to_arinc_queue). This routine then calls the resend_queues routine to ship all queued messages. This routine also assigns the printer sequence number.

AUTO_DISTRIBUTE_ACK_RECEIVED

This procedure extracts the sequence number from the received acknowledgment and, by comparing the site and class from the message, attempts to match the acknowledgment to a queued entry. There is a special test for wildcards, i.e., if the queue wildcarded the node identifier, any node with its site and class is considered a match. The specified queue is then searched (the in use bit must be set) looking to match the sequence number. If the sequence number matches, the entry is deleted. Any match or mismatch is logged into the hourly log files.

GET_WORD_LOCAL

This routine performs the get_word function on 1024 byte words.

OPEN_LOG_FILE

This routine closes any existing hourly log file and reopens a new one with the name:

</etms5>/etms_data/arinc/print.yymmddhhmmss

or

</etms5>/etms_data/nadin/print.yymmddhhmmss

The yymmddhhmmss is year/month/day/hour/minute/second timestamp.

CHECK_TIMER

This routine checks all timers.

Arinc.sprinter closes and open up another log file once an hour. *Arinc.sprinter* resends queued entries every two minutes. *Arinc.sprinter* polls itself every five minutes (message code is 16437). If the poll is not received back from node.sw within 5 minutes, the connection to node.sw is closed and then reopened. This checks the integrity of the node.sw connection.

CREATE_ARINC_QUEUE

This routine creates the queue files. The files are named: /etms_data/arinc/sprintq_file_xx or /etms_data/nadin/sprintq_file_xx where xx is a serial number. The *in use* bit mask is then cleared.

INSERT_TEXT_INTO_QUEUE

This procedure frames notifications internally generated by *Arinc.sprinter* and then calls arinc_queue_and_send to send the messages to all *Arinc.rprinter* modules.

INITIALIZE

This procedure initializes the *Arinc.sprinter* module. It performs the following steps:

(1) calls get_etms_path and terminates if it fails

(2) checks if the first argument is either *arinc* or *nadin*; if not specified, it defaults to *arinc*

(3) creates the directory structure and then creates its trace file either

</etms5>/arinc/trace/arinc.sprinter

or

</etms5>/nadin/trace/nadin.sprinter

(4) calls open_log_file to create the hourly log file

(5) sets all timers

(6) clears all queues in use masks

(7) reads the adaptation file and creates all queue files

(8) connects to the ETMS message switching system

(9) queues a *hub site queue manager restarted* message to all *Arinc.rprinters*

NET_OPEN_UP

Opens the connection to the ETMS message switching system. If unsuccessful, it sets a one-minute timer to try again.

INCREMENT_RETURN_COUNT

Increments the counter that indicates that a message was returned to sender for the specified queue.

NET_POLL

This procedure looks for messages from the ETMS message switching system. All returned messages are counted (INCREMENT_RETURN_COUNT), and all other messages are handed off to (NET_PROCESS_MESSAGE).

NET_PROCESS_MESSAGE

This procedure routes received messages to the appropriate processing routine. S0, S2-S9 are given to process_sx. S1 is given to process_s1. A polling received message (from itself) clears the outstanding poll flag. A message receive acknowledgment (*Arinc.rprinter* received the message) calls auto_distribute_ack_received. A message from *Arinc.server*, which requires an acknowledgment (*Arinc* message) causes an acknowledgment to be sent and arinc_queue_and_send to be called. A reconfigure command voids all queues, loads the new adaptation file, causes new queue files to be created, and causes a *HUB Site Queue Manager Reconfigured* message to be posted to all new *Arinc.rprinter* modules.

NET_SEND

Sends a specified message to node.sw. If any error occurs, except for port filled, the connection is broken and a new connection is made to noder.sw.

PROCESS

Performs an infinite loop by checking timers, looking for messages and waiting for something to happen.

PROCESS_SX

Formats a statistical report of the current state of *Arinc.sprinter* in response to a net.mail S0 or S2-S9 message.

PROCESS_S1

Sends a test message to all *Arinc.rprinters*, as well as an acknowledgment, back to the requestor.

ADDRESS_GET_VALUE

Extracts an address element from a text string presumed to be (xxx.yyy.zzz), i.e. returns xxx, yyy or zzz.

READ_AUTO_DISTRIBUTE_ADDRESSES

Reads the addresses from the adaptation file. The end of the file or the word **END** or **end** marks the end of the list. There can be a maximum of 16 addresses.

READ_ADAPT_FILE

Opens the adaptation file specified in argument 1 to the module or the default:

</etms5>/arinc/config/arinc.sprinter.config

or

</etms5>/nadin/config/nadin.sprinter.config

and then calls READ_AUTO_DISTRIBUTE_ADDRESSES to read the addresses.

READ_ADAPT_FILE

This routine sets the default adaptation file name and then looks for an adaptation filename from argument two to the module. If the argument exists it is used, otherwise use the defaults.

This routine then opens the file, any file open error causes program termination error, calls read_auto_distribute_addresses, and closes the file.

MAILBOX_SEND_FAILURE

This routine attempts to send a queued message to the ETMS message switching system. If the entry is timed out, it is removed from the queue and a *not sent* flag is returned.

If the connection to node.sw does not exist, a *not sent* flag is returned.

This routine begins its actual work (not above background steps) by updating the queued entry's last send attempt time and then tries to send the message to the ETMS. If the send is unsuccessful for any reason a *not sent* flag is returned; otherwise, a *sent* flag is returned.

RESEND_QUEUES

This routine resets the sending timer and then checks each recipient's queue.

The *in use* bit is checked for each entry. If it is set all the following operations are performed:

(1) Read the entry from the queue file.

(2) If the sequence number is the deleted mask, skip the record.

(3) If the entry was last resent within two minutes, skip it.

(4) Call mailbox_send_failure to resend the entry.

SEND_POLL

Rearms the event counter, sets a sent flag, and sends a poll addressed to this program (module) to node.sw.

CLEANUP_HANDLER

This routine performs an orderly module termination sequence. This procedure is called whenever a signal is received by the process whether internally or externally generated.

MAIN

This is the entry point for this module. It utilizes ETMS API calls to arm event handlers and obtain module arguments.

The module then calls initialize to do the obvious and then process to do the primary function. If somehow process returns (it performs an infinite loop), the API call pgm_exit is made to terminate the module.

7.7.1.2 Source Code Organization

Arinc.sprinter uses the standard ETMS API suite. Otherwise the module consists of one program named *arinc.sprinter5.c.*

7.7.1.3 Building Instructions

```
#*********************************************************
#*********************************************************
#                    ETMS          VERSION          5          OPEN          SYSTEMS
#                                                  ARINC                       SUITE
#*********************************************************
#*********************************************************

#---------------------------------------------------
#              Directories              for              this              specific              software
#---------------------------------------------------
ROOT_DIR                        =                                                                  ..
SHARED_DIR                      =                                                $(ROOT_DIR)/shared
ARINC_DIR       = ../arinc.osc

#---------------------------------------------------
#              Where              to              find              PasANSI              stuff
#---------------------------------------------------
PASANSI_SRC_DIR     =                                                $(ROOT_DIR)/PasANSI
PASANSI_INC_DIR     = $(ROOT_DIR)/PasANSI

#---------------------------------------------------
#              Where              to              find              API              stuff
#---------------------------------------------------
API_ROOT_DIR                    =                                                $(ROOT_DIR)/api
API_SRC_DIR                     =                          $(API_ROOT_DIR)/sources/api_openlib
API_INC_DIR                     =                                                $(API_SRC_DIR)
API_LIB_DIR                     =                                                $(API_SRC_DIR)
API_LIB                         =                                                      opensys
API_LIB_FULLNAME    = $(API_LIB_DIR)/lib$(API_LIB).a

#---------------------------------------------------
#              Compiler              flags              and              API              lib
#---------------------------------------------------
CFLAGS                                  =       -O      -I$(ARINC_DIR)      -I$(SHARED_DIR)      -
I$(API_INC_DIR)                                                          -I$(PASANSI_INC_DIR)
LIBS                                    = -L$(API_LIB_DIR) -l$(API_LIB)
```

```
#-------------------------------------------------
#                   Dependent                   include                   files
#-------------------------------------------------
INCLUDES                =       $(SHARED_DIR)/etms.lib.h        \
                                                $(PASANSI_INC_DIR)/PasANSI.h

#-------------------------------------------------
#                   Dependent                   object                    files
#-------------------------------------------------
OBJECTS                 =       $(SHARED_DIR)/etms.lib.o
\
                                                $(PASANSI_SRC_DIR)/PasFile.o
#-------------------------------------------------
#                     What                      to                        build
#-------------------------------------------------
all:                    arinc.qprinter arinc.rprinter arinc.sprinter

#-------------------------------------------------
#                   How                 to                build                 it
#-------------------------------------------------
#---------------------------------

arinc.qprinter:         $(OBJECTS)                      \
        $(ARINC_DIR)/arinc.qprinter.o                                           \
                                $(API_LIB_FULLNAME)
                @echo                           '[linking                       $@]'
                cc -o           $(ARINC_DIR)/arinc.qprinter             \
        $(ARINC_DIR)/arinc.qprinter.o   \
                                $(OBJECTS)                              \
                                $(LIBS)

$(ARINC_DIR)/arinc.qprinter.o:          $(ARINC_DIR)/arinc.qprinter.c   $(INCLUDES)
                @echo                           '[compiling                     $@]'
                $(CC) $(CFLAGS) -D ARINC -c  $(ARINC_DIR)/arinc.qprinter.c

#---------------------------------

arinc.rprinter:         $(OBJECTS)                      \
        $(ARINC_DIR)/arinc.rprinter5.o                                          \
                                $(API_LIB_FULLNAME)
                @echo                           '[linking                       $@]'
                cc -o           $(ARINC_DIR)/arinc.rprinter5            \
        $(ARINC_DIR)/arinc.rprinter5.o  \
                                $(OBJECTS)                              \
                                $(LIBS)

$(ARINC_DIR)/arinc.rprinter5.o:         $(ARINC_DIR)/arinc.rprinter5.c  $(INCLUDES)
                @echo                           '[compiling                     $@]'
                $(CC) $(CFLAGS) -D ARINC -c  $(ARINC_DIR)/arinc.rprinter5.c

#---------------------------------
```

```
arinc.sprinter:		$(OBJECTS)		\
	$(ARINC_DIR)/arinc.sprinter5.o		\
		$(API_LIB_FULLNAME)
	@echo		'[linking		$@]'
	cc -o		$(ARINC_DIR)/arinc.sprinter5		\
	$(ARINC_DIR)/arinc.sprinter5.o		\
		$(OBJECTS)		\
		$(LIBS)

$(ARINC_DIR)/arinc.sprinter5.o:		$(ARINC_DIR)/arinc.sprinter5.c		$(INCLUDES)
	@echo		'[compiling		$@]'
	$(CC) $(CFLAGS) -D ARINC -c  $(ARINC_DIR)/arinc.sprinter5.c
```

7.7.1.4 Constants

```
#define	CT_OUTPUT_MESSAGE_CODE		16435
#define		CT_OUTPUT_MESSAGE_ACCEPT_CODE		16436
#define	POLLING_MESSAGE_CODE		16437
#define MINUTES_BETWEEN_POLLS	5

#define		MAX_AUTO_DISTRIBUTE		16
#define		TWO_MINUTES		120
#define		FIVE_MINUTES		300
#define NUMBER_QUEUE_ENTRIES	128	/* IF THIS IS CHANGED, CHANGE THE DEFINITIONS OF
		BIT_MASK_128_T	which	uses	six	32	bit	integers	to
		map	which	records	are	in	use	*/
#define	MESSAGE_TIMEOUT_INTERVAL	(3600*6*4)	/*	6	hours	*/
#define	MAILBOX_CLOSE_TIMEOUT	(6*60)	/*	6	minutes	*/
#define		NET_EC		0
#define		TIME_EC		1
#define		MAX_EC		2
#define	WAIT_ON_OPEN	(1*60)	/* wait 1 minutes between retrying the open */
#define		LF		((char)10)
#define		_LF		{	(char)10,	0	}
#define		CR		((char)13)
#define		_CR		{	(char)13,	0	}
#define		ERROR_STREAM		stdout
#define NET_NIL	(-1)
```

7.7.1.5 Sprinter Data Structures

```
typedef		char		char3_t[3];
typedef		char		char4_t[4];
typedef		char		char6_t[6];
typedef		char		char8_t[8];
typedef		char		char10_t[10];
typedef		char		char32_t[32];
typedef		char		char128_t[128];
typedef		char		char256_t[256];
```

```
typedef                 char                                    char200_t[200];
typedef                 char                                    char512_t[512];
typedef                char                                   char1024_t[1024];
typedef                char                                   char6000_t[6000];
typedef                char                                   char8192_t[8192];
typedef                char                                   char8100_t[8100];
typedef short   int3_t[3];

typedef
struct                                                          control_format_t
{
   INT32                                                                seq;
   CALTIME                                                             time;
   short                                                           data_size;
   char6000_t                                                           data;
}          control_format_t;

typedef
struct                                                                queue_rec_t
{
   INT32                                          timeout,     last_sent,     seq;
   control_format_t                                                     msg;
}          queue_rec_t;

typedef
struct                                                              queue_control_t
{
   INT32                                             fill,     empty,     count;
   control_format_t                                                     msg;
}          queue_control_t;
```

7.7.2 Rprinter Module

Purpose

The *Rprinter* module or *Arinc.rprinter* receives messages from the *Arinc.sprinter* module and inserts the messages into a shared memory region that *Arinc.qprinter* can then use to place the messages onto the printer. *Arinc.rprinter* also logs all messages into hourly and individual log files.

Design Issue: Environment

Arinc.rprinter is compliant with ETMS version 5 network addressing and open systems. It has no hardware requirements.

Execution Control

Initialization. The module initializes by calling the initialize procedure, which is described below in detail. The important parts of initialization is to determine if it is running in *Arinc* or *Nadin* mode (third argument). The initialize procedure then creates the two shared regions, one for all received messages, the other for non SI type messages. The initialization phase is completed by creating the log file and connecting to ETMS message switching.

Termination. The termination of *Arinc.rprinter* is handled by a routine called cleanup_handler. Cleanup_handler is called by the operating system when signals are received; these signals can be internally generated or externally generated. The cleanup_handler routine unmaps the shared regions, closes the ETMS connection, and closes out the log files.

Input

Arguments. *Arinc.rprinter* requires three arguments in its command line. The first argument must be the name of the shared memory region that receives only SI type messages. The second argument is the name of the shared memory region that contains all received messages. The third argument is a word that is either *arinc* or *nadin* (the *Arinc* or *Nadin* flag). If neither word is specified, *Arinc.rprinter* defaults to being in the *Arinc* mode.

Messages From Arinc.sprinter. The *Arinc.sprinter* must have the address of *Arinc.rprinter* (and *Nadin.sprinter* must have the address of *Nadin.rprinter*) in its configuration file in the auto-configure section. Otherwise no **Arinc** (or **Nadin**) messages will be printed. All messages from *Arinc.sprinter* must have a message code of 16435 and the first four data bytes are presumed to contain the *Arinc* message sequence number and the subsequent four having the *Arinc.sprinter* assigned printer sequence number. *Arinc.rprinter* then sends the same message back to *Arinc.sprinter* with a message code of 16436 as an acknowledgment.

There is a two-step filtering of *Arinc* messages in *Arinc.rprinter*. All received messages are determined to be SI type or not. All messages go into one shared region (second argument to this module). Only non SI type messages go into the non SI shared memory region (first argument to this module). The determination of an SI message is:

(1) Skip over two lines of text (line feed marks end of each line).

(2) If the next word is `****' it might be the second part of a multiple-part message. The subsequent word must be PART and the following line must be ACID DEP DEPT EDCT CTA, where ACID is 2-7 characters with first character being a letter, where DEP is 3-4 characters, and DEPT/EDCT/CTA must each be a letter followed by 4 digits. If all these rules apply, this is considered to be an SI.

(3) If the first word after skipping the two lines is SI, it is considered to be an SI.

(4) If the first word after skipping the two lines is FLIGHTS, it is considered to be an SI (FLIGHTS REVISED BY SI).

(5) If the first three words after skipping the two lines are:
CONTROLLED FLIGHTS FOR
CANCELLED FLIGHTS FOR
OPEN SLOTS FOR
the message is considered to be an SI.

Statistics Requests. *Arinc.rprinter* responds to *Net.mail* S0 and S1 requests. Net.mail S2-S9 requests are responded to with the same unsupported command error. Refer to the detailed descriptions of process_s0, process_s1 and process_sx for further details.

Output

Shared Region. The primary output of *Arinc.rprinter* is messages to the two shared regions. *Arinc.rprinter* determines if the message is an SI type and does not write it to one region but it writes all messages to the other shared region.

Log file handling. *Arinc.rprinter* writes out hourly log files which contain all received messages. The file is open, appended to and then closed for each message. The file is named either

/etms_data/arinc/print.yymmddhhmms

or

/etms_data/nadin/print.yymmddhhmmss

where yymmddhhmmss is the two-digit year, month, day, hour, minute, second.

Each input message is logged into its own file named

/etms_data/arinc/ddhhmm.saddress.yyddmmhhmmss.seq#

or

/etms_data/nadin/ddhhmm.saddress.yyddmmhhmmss.seq#.

Where ddhhmm is the embedded data-time group in the message, saddress is the source *Arinc* or *Nadin* address, yyddmmhhmmss is the same as the above file, and seq# is the printer sequence number.

Statistical Reports. A statistics report is returned to those users who send a net.mail S0-S9 to the *Arinc.rprinter* module. An S0 returns statistics, an S1 is an acknowledgment that a test message was inserted into the shared regions, and an S2-S9 return the information that S2-S9 are not supported by *Arinc.rprinter*.

Processing

The processing phase is one large infinite loop; the loop is gated by event counters on timers and messages from the ETMS. On each loop cycle, all timers are checked and all messages from ETMS are resolved. Refer to the *Process* routine for more details.

Arinc.rprinter receives unsolicited messages from *Arinc.sprinter* via the *Arinc.sprinter* auto-distribute function. The address of this *Arinc.rprinter* (either specific or wildcard) must be specified within the *Arinc.sprinter* adaptation file. *Arinc.rprinter* must acknowledge each message from *Arinc.sprinter* or *Arinc.sprinter* will continue to re-send each message.

Timed Events. *Arinc.rprinter* strobes a counter in each shared region every 9.5 seconds. *Arinc.rprinter* sends itself a poll through node.sw on a timed basis (every 5 minutes). If a response is not received by the next polling interval, the connection to node.sw is closed and re-opened. *Arinc.rprinter* opens a new log file each hour.

Shared Region. *Arinc.rprinter* maintains two shared memory queues implemented as disk files. The *Arinc.qprinter* modules read these shared regions and prints the contents. There is a limited amount of handshaking between *Arinc.rprinter* and *Arinc.qprinter*.

Error Conditions and Handling

Exceptional error handling causes entries to be made into the hourly log file. Fatal signals to the module cause an orderly shutdown by having the signal handler call the routine named cleanup_handler within *Arinc.rprinter*.

7.7.2.1 Rprinter Routines and Procedures

Add_to_region

This procedure moves a message into the shared region(s). If it is not an SI, it goes into both; an SI only goes into the one all inclusive region. The region is considered to be circular so that any old characters remaining in the regions are over-written. Any non-printing character other than tab/cr/lf is removed from the message as it is being moved into the shared region. The write count is incremented accordingly, a flag of 0xFF is appended to the region (to mark the end of the region), and a completed message counter is incremented (this is a pseudo event counter).

Get_word_local

This procedure extracts a 1024 byte word from the given buffer. Otherwise it is identical to the universal get_word.

Get_source

This procedure extracts either the *Arinc* or *Nadin* source address and date time group from a message.

The rules for *Arinc* are to skip over the priority (typically QU) and then look for **lf** followed by a period. The next word is the source address only if it is seven or eight bytes with an upper case letter as the first character. If the format does not match, seven xs are returned. The next word must be the date time group; if this is not six characters with the first character being a digit, **999999** is returned.

The rules for *Nadin* are to skip over the first linefeed. The next word must be the date time group, if this is not six characters with the first character being a digit **999999** is returned.

The next word is the source address only if it is seven or eight bytes with an upper case letter as the first character. If the format does not match, seven **x**s are returned.

Is_second_part

This routine looks to see if the message is the second or subsequent part of a long message. This is part of the *is this an SI* logic.

If the first word is not PART, return false (not second part). Skip over the rest of the line. The next line must be ACID DepAir DEPT EDCT CTA which results in the following rules:

(1) first word: two to seven characters, first one is an upper-case letter

(2) second word: three to four characters, first one is a upper-case letter

(3) third word: five characters, first is upper-case letter and second and last are digits

(4) fourth word: five characters, first is upper-case letter and second and last are digits

(5) fifth: five characters, first is upper-case letter and second and last are digits

If all the above rules match, this is a multi-part message (considered to be an SI).

Check_for_SI

Skip over two lines (two linefeeds). If the next word is exactly **** call is_second_part, which returns if it is an SI or not.

Otherwise the rules are:

(1) single word = SI -> is an SI.

(2) single word = FLIGHTS -> is an SI.

(3) three words = CONTROLLED FLIGHTS FOR -> is an SI.

(4) three words = CANCELLED FLIGHTS FOR -> is an SI.

(5) three words = OPEN SLOTS FOR -> is an SI.

If none of the above rules match, it is not an SI.

Format_header

This procedure formats a standard header that will be used in the trace files. This header contains the timestamp and the sequence numbers of the messages.

Cavort_with_valid_message

This routine processes a message from the *Arinc.sprinter*. This routine extracts the *Arinc* and the printer sequence number, sender's address, and the date time group from the message. This routine then calls check_for_si to determine if it is an SI message. When the above paperwork is done, this procedure formats the standard header by calling format_header, removes non-printing ASCII characters from the buffer and then calls write_to_sio to write the data into the log files and the shared regions. The message cannot exceed 8000 characters, but the code segments each message into 8000-byte blocks.

Format_init_header

This routine, called as part of the initialization process, places a module initialized message into the shared regions.

Initialize

This routine performs the module initialization. The following steps are performed by calling auxiliary procedures:

(1) Get_etms_path to determine the file system layout

(2) EVT_create to create the event handlers

(3) Determining if this module is in *Arinc* or *Nadin* mode (pgm_get_arg)

(4) Creates the output directories, either /etms_data/arinc or /etms_data/nadin

(5) Obtaining the names of the two shared regions and creating them (Make_region_owner)

(6) Open_log_file to create the hourly log file

(7) Net_open_up to connect to the ETMS messages switching system

Make_region_owner

This procedure creates a shared memory mapped file. This procedure then initializes the control information within the mapped memory to allow other processes (*Arinc.qprinter*) to synchronize their shared memory read operations with the write operations of this process.

Net_open_up

This procedure connects *Arinc.rprinter* to the ETMS message switching system. If there already is a connection, this routine closes it prior to attempting to open a new one. The connection class is either Arincp or Nadinp. If the internal call to the ETMS API net_open fails (and no connection is made), this routine sets a one-minute timer event counter.

Net_poll

This procedure looks for messages from the ETMS message switching system. If there is currently no connection, procedure net_open_up is called.

Otherwise this procedure is one infinite loop which is terminated by an error on a message read or the fact that there are no more messages (ETMS API net_get_msg does the actual message reading). Any read error (except no more data) causes the connection to be closed and reopened.

All returned messages are ignored. All other messages are passed onto net_process_message for resolution.

A statistics request from net.mail of S0 causes process_s0 to be called. A statistics request from net.mail of S1 causes process_s1 to be called. All other net.mail statistics requests (S2-S9) cause process_sx to be called.

A message sent by *arinc.rprinter* to itself clears the polling flag. If the flag is not cleared, this module will disconnect and re-connect to the ETMS message switching system.

A message from the *Arinc.sprinter* module with the guranteed delivery message code (16435) causes an immediate echo of the message with the acknowledgment flag (16436) set. The received message is then passed onto cavort_with_valid_message for resolution. All other message codes are totally ignored.

Net_send

This procedure is a universal transmitter. All messages from this module to anywhere in the ETMS utilize this procedure. Any sending errors (outside of port filled/no room in channel) will cause this procedure to disconnect and re-connect to the ETMS message switching system.

Open_log_file

This procedure creates the hourly log file. It then sets a one-hour timer to allow the next file to be created on the hour.

Reopen_log_file

This routine will close and re-open the existing hourly log file in append mode. This allows the UNIX tail and cat functions to operate properly on some UNIX platforms.

Process

This is the main processing loop of *Arinc.rprinter*. This routine is an infinite loop. The loop performs the following operations:

(1) Increments a heartbeat counter in each shared region on a scheduled basis (approximately every 10 seconds). This allows *Arinc.qprinter* modules to sense that they are not properly connected to the shared region.

(2) Sends a poll to itself on a scheduled basis. If the timer expires and there has been no receipt of the previous poll, this routine disconnects and re-connects to the ETMS message switching system.

(3) Checks to see if it is time for another hourly file to be created

(4) Re-arms the quarter-second timer

(5) Looks for messages from the ETMS message switching system

Dump_region_info

This routine formats the specified shared region's control information into a supplied buffer. This is intended to be part of the statistical reports that users can request.

Process_S0

This procedure supplies statistical information about *Arinc.rprinter* operation to users in response to a net.mail S0 request.

Process_S1

This procedure inserts a test message into both shared regions which should be read by all connected *Arinc.qprinter*s. This is performed in response to a net.mail S1 request. The requesting net.mail receives an acknowledgment of the request.

Process_sx

This procedure sends a *request unsupported* message to any user who sends this module a net.mail S2 to S9 request.

Send_poll

This procedure sends a message addressed to itself to the ETMS message switching system. If the message is not received by the next polling interval, the connection to the ETMS is deemed non-functional and is terminated (and re-opened).

Write_to_sio

This routine performs the main message output in this module. It writes the message to either or both shared regions, depending upon whether it is an SI or not. It creates a file for each message, writes the message, and closes the file. This routine also writes the message to the hourly log file, closes it, and then reopens it. (Refer to reopen_log_file for details.)

Cleanup_handler

This procedure is called at module termination to deallocate all resources. All open files are closed, the shared memory files are unmapped, and the connection to ETMS is severed.

MAIN

This is the entry point for this module. It utilizes ETMS API calls to arm event handlers and obtain module arguments.

The module then calls *initialize* to start, and then *process* to perform the primary function. If *process* returns (it performs an infinite loop), the API call pgm_exit is made to terminate the module.

7.7.2.2 Source Code Organization

Arinc.rprinter uses the standard ETMS API suite. Otherwise the module consists of one program named arinc.*sprinter5.c* with a shared-region-insert filename *arinc_q_region.h.*

7.7.2.3 Building Instructions

Refer to the *Arinc.sprinter5.c* section for the Makefile.

7.7.2.4 Constants

```
/*                              EVENT                           COUNTERS                        */
#define                                 NET_EC                                                  0
#define                                             TIME_EC                                     1
#define NUM_EC  2

#define                 NET_NIL                                                                 (-1)
#define                                             STX                                         ((char)2)
#define             _STX                                        {           (char)2,        0       }
#define                                 CR                                                      ((char)13)
#define                 _CR             {               (char)13,               0               }
#define                                 LF                                                      ((char)10)
#define                 _LF             {               (char)10,               0               }
#define  TEN_SECONDS  (10*4)            /*  10  SECONDS  -  USED  FOR  MBX_$TIMED_OPENS  */
#define ONE_MINUTE  (1*60*4)    /* WAIT 1 MINUTE BETWEEN RETRYING THE OPEN */

#define                         CT_OUTPUT_MESSAGE_CODE                                          16435
#define                 CT_OUTPUT_MESSAGE_ACCEPT_CODE                                           16436
#define       POLLING_MESSAGE_CODE                                                              16437
#define MINUTES_BETWEEN_POLLS         5

                /*      allocate        16      *       64K     bytes   of      buffer  space   */
#define REGION_BUFFER_SIZE  0xfffff
```

7.7.2.5 Rprinter Data Structures

```
typedef                 char                                                    char2_t[2];
typedef                 char                                                    char3_t[3];
typedef                 char                                                    char4_t[4];
typedef                 char                                                    char5_t[5];
typedef                 char                                                    char6_t[6];
typedef                 char                                                    char7_t[7];
typedef                 char                                                    char8_t[8];
typedef                 char                                                    char9_t[9];
typedef                char                                                    char10_t[10];
typedef                char                                                    char32_t[32];
typedef               char                                                    char256_t[256];
typedef              char                                                    char1024_t[1024];
typedef              char                                                    char6000_t[6000];
typedef              char                                                    char8100_t[8100];
typedef char    char8192_t[8192];

typedef
struct                                                                  control_format_t
{
   INT32                                                                        seq;
   CALTIME                                                                      time;
   short                                                                        data_size;
```

```
    char6000_t                                                    data;
}             control_format_t;

typedef
    struct                                                        region_t
    {
      INT32    heartbeat;       /* incremented every 10 seconds - if no increment -> clients should reconnect */
      INT32         rprinter_pid;          /* process ID of rprinter - qprinter reopens if it changes*/
      INT32    event_counter;    /* incremented whenever an entry is written to the region - performs no process
signalling                                   !!!                                          */
      INT32              write_count;                    /* total bytes written to the queue */
      INT32              read_count    [8];         /* how many bytes read from this channel */
      INT32              fill;                                        /* next byte to place data */
      INT32              empty    [8];                              /* qprinter programs utilize this */
          short     qprinter_pid  [8];        /* != 0 means someone connected at some time, they may not have
disconnected                                  though...                                   */
      short          valid;                          /* =0 region is o.k., <> 0 then don't use */
      char                                          buffer[REGION_BUFFER_SIZE        +          1];
    }   region_t;

typedef region_t    *region_ptr_t;
```

7.7.3 Qprinter Module

Purpose

The *Qprinter* module or *Arinc.qprinter* is responsible for placing the data from the *Arinc* (or *Nadin*) network onto a piece of paper (or re-directed to another application).

Execution Control

Initialization. The initialization phase is concerned with reading in the module arguments, connecting to the shared memory region, and configuring the serial port.

Termination. The termination phase is concerned with the orderly shutdown of the application so that all allocated resources are properly released.

Input

Arguments. *Arinc.qprinter* accepts four module arguments. The first argument is the name of the shared memory region. The shared memory region must exist on the same node that is running both *Arinc.rprinter* and *Arinc.qprinter*. The second argument is the serial port number. This must later be converted to the name of the printer device port. The third argument is used as an index. This index signals whether this module will update global parameters in the shared region or not. A positive value of 1-8 means to update the control information for a specific control set. A value of zero means to not update the control

information. The fourth argument is the speed of the serial port in bps; suggested values are 4800, 9600, 19200, or 38400.

Output

Printouts. The output from this module is text to a serial port or to the standard output stream.

Shared Region. The shared region is updated only if the third module argument indicates the specific control table to update.

Statistical Reports. This module does not interact with the ETMS message switching system. Most status messages are written to the serial port.

Trace File. This module logs all module heartbeat failures and subsequent shared memory region disconnects into a trace file that is always less than 32,767 bytes (the file is erased whenever it reaches this limit). The name of the trace file is /etms_data/arinc/qprinter_log.PID or /etms_data/nadin/qprinter_log.PID, where PID is the UNIX program identifier.

Processing

Processing is one loop of waiting for characters to appear in the shared region and then transferring them to the serial port.

Timed Events. The only timed event in use is to check whether the shared region heartbeat is being incremented. If it is not, *Arinc.qprinter* closes and re-opens the shared memory region.

Shared Region. All input to this module is via the shared memory region. This region is a circular byte buffer. *Arinc.qprinter* moves characters, not messages, from this buffer directly to the serial port. The shared region also has a suite of control fields, which includes a region valid flag, a heartbeat counter, a next byte to fill pointer, and a set of eight control fields, which include the process identifier and the next byte to read.

Error Conditions and Handling

This module re-establishes communications with the shared memory region if the in-use flag is cleared or a heartbeat counter is not updated in a timely fashion.

Most other errors cause module termination.

7.7.3.1 Qprinter Routines and Procedures

Log_error

This routine logs all heartbeat timeouts into the trace file. The file is opened, written and then closed on each event.

Get_from_region

This routine looks for characters from the shared memory region. It returns a negative byte count if the region is not mapped or is marked as being invalid.

This routine moves at most 8000 bytes from the shared region. Non-printing characters are not moved (carriage return, linefeed, tab are considered printable characters only in this context). The move from the shared region is completed when 8000 bytes are removed or a flag byte of 0xFF is found or the *next byte to write* pointer is reached.

The shared memory region control area is updated only if the third module argument is in the range of one to eight.

Make_region_user

This procedure connects the shared memory region maintained by *Arinc.rprinter* to *Arinc.qprinter*. The first step performed is to write that a connection attempt is being made to the printer. If the region is currently connected, disconnect the region.

If the connection has failed or is marked as invalid, write an error to the printer, wait a discrete interval, and then return failure to the calling module.

On a successful connection, write a successful connection message to the log file and to the printer and then update the shared memory region and the environmental variables.

Initialize

This routine configures the *Arinc.qprinter* module. The first step that is performed is to read the shared memory region name from the first argument. The shared memory name is scanned and if the text *Nadin* appears, set the *Nadin* flag.

The next step that is performed is to create the log file name as /etms_data/arinc/qprinter_log.PID or /etms_data/nadin/qprinter_log.PID. This routine then calls open_sio to set up the serial port.

The *Arinc.qprinter* then goes into a gated loop trying to connect to the *Arinc.rprinter* shared memory region (make_region_user).

The third argument to the program is then used to determine if the shared region should be updated. A value of 0 to 7 means to update the appropriate shared memory control area. A negative sign in front of the value means to start printing from the beginning of the shared memory region. Otherwise printing will start on the next addition to the shared memory region.

Open_sio

This routine defaults the print device to /dev/tty00. The second module argument is then checked for the range of 1 to 3. The last character of /dev/tty00 is then overlayed; if the argument is 2, the device name is /dev/tty02. A negative sign on the second program argument means echo all output to the standard output stream.

The serial port is then opened using standard POSIX calls, and program argument 4 is evaluated for line speed. If the line speed is not valid (see the above input section) it is defaulted to 9600 bps. The speed of the serial port is then set, a printer is started, and a printer is *starting* message is output to the printer (write_to_sio).

Process

This procedure performs the main programming loop. It creates the time event counter and then begins an infinite loop. If the shared region heartbeat is not incremented in a timely fashion, make_region_owner is called to disconnect and re-connect to the *Arinc.rprinter* shared memory region.

The inifinite loop is gated by a 1/4 second event counter. A call to Get_from_region is made on each 1/4 second interval. A negative byte count causes a loop to make_region_user with its error message to the printer until a connection is re-established to the *Arinc.rprinter* shared region. A positive byte count is passed to write_to_sio to output the data.

Write_done

This routine is used at module termination to write a module termination message to the printer.

Write_to_sio

If the echo to standard output flag is set, this module uses the UNIX write call to write to the standard output.

Otherwise this routine uses the UNIX write call to write to the serial port.

Cleanup_handler

This routine shuts down the *Arinc.qprinter* module. It calls the write_done procedure, disconnects from the shared memory region, closes the serial port, and exits.

Main

The main routine uses ETMS API calls to set a signal handler and obtain the original module arguments. It then calls the *INITIALIZE* and the *PROCESS* routines.

7.7.3.2 Source Code Organization

This module is composed of one source code program.

7.7.3.3 Building Instructions

Refer to the *Arinc.sprinter* module for the Makefile.

7.7.3.4 Constants

Refer to the *Arinc.sprinter* module for the constants used by this module.

7.7.3.5 Qprinter Data Structures

Refer to the *Arinc.sprinter* module for the common data structures used by these modules.